17473

NATIONAL LIBRARY
OTTAWA

CANADA

BIBLIOTHÈQUE NATIONALE
OTTAWA

NAME OF AUTHOR.....WAYNE...KENNETH..CHELAK.....

TITLE OF THESIS....PROGRAM...REORGANIZATION.....

...FOR..PERFORMANCE..IMPROVEMENT

...IN...PAGING..SYSTEMS.....

UNIVERSITY.....ALBERTA.....

DEGREE FOR WHICH THESIS WAS PRESENTED....M.S.....

YEAR THIS DEGREE GRANTED.....1973.....

Permission is hereby granted to THE NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed)...W. K. Chelak...

PERMANENT ADDRESS:

#1512 - 8210 - 111 St

Edmonton, Alta.

DATED....Nov. 1.....19 73

NL-91 (10-68)

THE UNIVERSITY OF ALBERTA

PROGRAM REORGANIZATION FOR PERFORMANCE IMPROVEMENT IN PAGING SYSTEMS

by

(C) WAYNE K. CHELAK

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT of COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1973

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled PROGRAM REORGANIZATION FOR PERFORMANCE IMPROVEMENT IN PAGING SYSTEMS submitted by WAYNE K. CHELAK in partial fulfilment of the requirements for the degree of Master of Science.

T.A. Marsland
.....
Supervisor

Barry J. Mailloux
.....

Senna
.....

Date... Oct. 5, 1973...

ABSTRACT

The increasing popularity of large virtual-memory computing systems has provided the stimulus for new research in the areas of computer-system performance evaluation and program behaviour. The focal point of this study involves a possible technique for improving program performance in paged-memory systems. Attention is focused on optimization of the behaviour of individual programs to eliminate excessive paging. The purpose of this research is to investigate whether or not a reorganization of the relocatable modules of the program within the pages is indeed a viable procedure. In order to achieve a restructuring of the program's pages, information concerning the interaction of the program modules is necessary. As a result, the experimental work includes monitoring the behaviour of several selected programs as well as examining the benefits obtained from a page reorganization.

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to Professor T.A. Marsland, my supervisor, for his advice, criticism, and guidance throughout the duration of the research and the preparation of the thesis.

The financial assistance provided by the National Research Council of Canada, in the form of scholarships, and the Department of Computing Science, in the form of teaching assistantships, is gratefully acknowledged.

TABLE OF CONTENTS

CHAPTER			PAGE
1.	INTRODUCTION		1
	1.1	Introduction	1
	1.2	Single Program Systems	3
	1.3	Early Supervisory Systems	5
	1.4	Multiprogramming Systems	5
	1.5	Virtual Memory Systems	8
		1.5.1 Addressing	10
		1.5.2 Implementations	11
		1.5.2.1 Segmentation	13
		1.5.2.2 Paging	14
		1.5.2.3 Segmentation and Paging	17
		1.5.3 Additional Considerations	20
2.	PAGING		21
	2.1	Introduction	21
	2.2	Paging Policies	22
	2.3	Page Replacement Algorithms	25
		2.3.1 RAND - Random Replacement	27
		2.3.2 FIFO - First In / First Out Replacement	28
		2.3.3 LRU - Least Recently Used Replacement Replacement	28
		2.3.4 Working Set Algorithm	30

2.3.5 Optimal Replacement..... 31
2.3.6 Adaptive Replacement..... 32
2.4 Conclusion..... 34

3. PROGRAM BEHAVIOUR IN A PAGING ENVIRONMENT..... 35
3.1 Introduction..... 35
3.2 Representation of Program Behaviour..... 36
3.3 Gauging Program Behaviour..... 40
3.4 Monitoring Program Execution..... 45

4. RESTRUCTURING PAGE CONTENTS..... 51
4.1 Introduction..... 51
4.2 Assignment of Program Code to Pages..... 51
4.2.1 Pagination to Reduce Internal Fragmentation..... 53
4.2.2 Pagination to Reduce Inter-Page Referencing..... 53
4.3 Optimal Sector Placement..... 55
4.4 Heuristic Sector Placement Algorithms..... 60
4.5 Practicality of Page Restructuring..... 63

5. ASSESSMENT OF STATIC PROGRAM REORGANIZATION... 65
5.1 Introduction..... 65
5.2 Monitor Results..... 67
5.3 Program Reorganization..... 72

5.4 Interpretation of Program Reorganization Statistics..... 78

6. CONCLUSIONS..... 80

BIBLIOGRAPHY..... 88

APPENDIX..... 94

LIST OF TABLES

TABLE		PAGE
1	Program Instruction Types.....	70
2	Program Reference Characteristics.....	71
3	Potential Page Faults.....	73
4	Potential Page Faults: Program 1 Reorganized According to Run 1b.....	74
5	Potential Page Faults: Considering Super Pages.	76
6	Potential Page Faults: Ignoring Dynamically Acquired Regions.....	77
7	Potential Page Faults: Ignoring Dynamically Acquired Regions and Considering Super Pages...	77

LIST OF FIGURES

FIGURE		PAGE
1	Basic Memory Mapping Scheme.....	12
2	Memory Mapping for Segmentation and Paging.....	18
3	Graphical Representation of Program Structure..	37
4	Boolean Connectivity Matrix.....	38
5	Transition Probability Matrix.....	38
6	Inter-Sector Reference Matrix.....	69

CHAPTER 1

INTRODUCTION

1.1 Introduction

The emergence of large complicated computer systems in which several problem programs compete for the resources of the system has led to increased interest in the areas of computer systems performance evaluation and optimization. The operating system has come to assume the responsibility of managing the job stream presented to the CPU for processing and that of maintaining full usage of the hardware resources.

The desire to maintain balanced hardware resource usage led to the concept of processing several jobs simultaneously; however, the idea depends on the availability of a balanced workload to utilize all resources equally. If several jobs attempt to monopolize a particular resource then one will eventually hinder the execution of them all. This study is aimed at reducing the impact of individual jobs on other jobs and the operating system, specifically a paging operating system.

The nature of this research is purely exploratory and is concerned with the optimization of programs executing in a paging operating system. The proposed scheme involves reordering or reorganizing the relocatable modules of a program so that it is executed more efficiently.

The efficiency of execution is judged to be dependent on the system overhead required to handle the program. In particular, for paging systems, the goal is to reduce the number of program interrupts caused when the system must bring needed data from auxiliary storage to main memory. It is the main memory resource which is in great demand in a heavily loaded paging system and it is desirable to reduce the contention for main storage.

Since the research is at the stage of investigation, it must draw knowledge from related areas, including program behaviour, optimization techniques and operating systems. By gathering experimental data from programs running in a paging system and examining the proposed reorganization it is hoped that the practicality of such a scheme can be established. Also, from the information collected and the way in which it was collected, new perspectives on program monitoring can be gained. The statistics which have been gathered prove to be extremely useful in evaluating programming techniques for paging systems.

In the remainder of this introductory chapter, the development of computer memory systems is explored. The review concentrates on the organization of such systems rather than the particular hardware devices. In addition, methods of addressing information are discussed [A3, D4]. The review is presented on the basis of complexity, sophistication, and for the most part, chronological order.

However, it is difficult to present the exact time line of events, since the implementation of various systems did not parallel the sophistication of the systems.

1.2 Single Program Systems

Since the advent of the stored-program concept, introduced by von Neumann [V1] and first implemented on the Edvac and Edsac computers, a prime consideration in the development of computer systems has been the allocation of storage facilities. In the earliest computer systems, this problem was left entirely to the user. There existed no operating system and few utility programs so the programmer was responsible for loading and running his own programs from the main console. The entire memory was free to be occupied by the user program; therefore, the only problem was ensuring that the program would fit into memory.

As the methods of storage allocation have developed from the early single-job environment to the present multiprogramming and timesharing systems, the method of addressing information has also evolved. This evolution has been in respect to the translation of a program's name space (ie. set of names used in the program to refer to information items) to the computer's physical address space (ie. set of physical address locations in the computer's memory) [R1]. The early programs, written in machine language, required all references to be physical memory

addresses. Thus the association of the name space to the address space was performed by the programmer. The development of the assembler-loader allowed the programmer to employ symbolic addresses which were converted to absolute machine addresses at assembly time.

The occurrence of situations in which a program grew larger than the available memory led to the development of overlay techniques. The program is manually divided into segments such that each is capable of fitting into main memory (ie. that memory which is directly addressable by the processor). As each segment terminates execution (ie. calls another segment) it is overlayed by its successor, which is located on some backing storage device, and execution resumes. Usually a block of memory, known as a common region, is reserved for storage of data being transferred from one program segment to the next. A more general approach to the problem of programs which can be only partially contained in main memory is referred to as "folding" [S1]. The name space of the program must be rearranged to fit into the smaller address space, and references to information outside the main memory must result in the fetching of the information from some backing storage device.

1.3 Early Supervisory Systems

It soon became evident that unless computing systems offered ease of access their use would never gain popularity. The solution chosen was to write software to handle the cumbersome and recurring tasks such as loading programs, handling interrupts, converting codes and even recognizing errors. The nucleus of supervisory programs is usually resident in memory and thus the main expense is the loss of memory capacity for the user program. In this two level system, consisting of supervisory programs on one level and user programs on another, storage allocation was still basically left to the user, although there were instances where the supervisory programs performed overlays on command. The programmer was still obliged to segment the program and insert commands to request the overlays.

1.4 Multiprogramming Systems

Multiprogramming was the next step in the desire to obtain the most efficient system for a given hardware configuration. This concept requires the co-existence of several programs in working storage, and utilization of as much of the hardware resources as possible. Since any system is normally limited by its most overworked component, the goal is to balance the workload so that all resources are kept busy but none are overly busy. The development of the data channel and interrupt facility, which allowed I/O

transfers and instruction processing to proceed in parallel, was a step towards total resource utilization. The CPU can perform useful work on one program while another is awaiting the completion of an I/O operation. The problems associated with such a system are complexity, resource scheduling and protection; however, the results as measured by total throughput can be profitable.

With the introduction of the relocation register [K7, C2] the task of associating the name space to the address space was left to the loader. A symbolic address is translated not into an absolute memory location but rather into a displacement from the program's origin. The address at which the program is loaded is placed in the relocation register, and during execution the actual memory address of any reference is calculated by summing the contents of the relocation register and the relative address. The evolution of these "a priori allocation algorithms" [B3] has thus deferred the translation of program references to physical addresses from program preparation time, to assembly time, to load time.

An important consideration is that of allocating main memory to the programs sharing the system. One simple technique, exemplified by IBM's OS/360 MFT system [I3], is that of partitioning memory and allowing only one program within each partition. The program remains there until its completion; however, it receives CPU service for only a

fraction of the elapsed time. CPU service is usually transferred from one of the resident programs to another, either after a fixed timeslice of execution, upon an I/O request or an interrupt.

A slight variation of the partitioned-memory system allows a user program to be "rolled out" (ie. swapped onto some backing storage device) of its partition when it is temporarily blocked and a new program to be rolled into that partition. The roll in/roll out feature would also be applicable to systems where all jobs are assigned priorities; here, a job with high priority entering the system would cause a job in memory with lower priority to be rolled out.

An obvious shortcoming of the fixed-partition memory system is that of utilizing the memory capacity to its full extent. The unused memory in each partition must remain wasted for the life of the program within the partition. Thus the logical consequence is to allocate storage to the program only as required. For example, IBM's OS/360 MVT [I4] allocates in blocks of 2K bytes the requested space for the duration of a job step. Memory is thus occupied with several jobs of various sizes residing in contiguous regions. A task can be initiated only if there is sufficient contiguous storage to meet the request. The storage assignment is made on the basis of the first available storage area from the high end of the user memory

region large enough to hold the task. If the task is smaller than an available region of processor storage the unused portion forms a new partition. However, if all available partitions are too small to accommodate any incoming task they will remain unused until two neighbouring partitions become available and are merged. No new task will be initiated until the merging process creates an available partition of sufficient size.

1.5 Virtual Memory Systems

Storage devices can usually be ranked according to their access rate and cost. Since a higher access rate often implies a higher cost per bit of storage, the devices which can be accessed faster generally have less overall capacity. As a result, it was never economically feasible to implement a system with substantially unlimited memory directly addressable by the processor. Therefore, the use of an auxiliary memory to hold information until actually required in main memory by the processor has been employed with the intention of increasing the effective memory size.

A problem associated with these systems is the translation of program references into memory addresses. One solution requires that the programmer preplan the segmentation of his programs and data areas and then use nonautomatic or semiautomatic overlays [P1]. The other solution involves an operating system which decides how

programs and data areas are to be segmented and then automatically handles all overlays. It is this second solution which has resulted in the development of virtual-memory systems [D1].

In a virtual-memory system, programs can be written with little regard for real-memory limitations. The hardware/software system is then responsible for segmenting the program and transferring the segments between main and auxiliary memories at the required times. A reference to some location not within the bounds of the current segment may be resolved in one of two ways. If the segment containing the desired information is in main memory, the reference is resolved with a physical address. If however, the information is in a segment located in auxiliary memory then the system must move the segment into main memory and determine a main memory address for the information.

Virtual-memory systems are usually implemented through a combination of hardware and software rather than just software, in order to be effective. The segments may be of fixed or variable length, depending on the particular implementation. Loading of segments is performed according to the dynamic situation with respect to the program and other concurrently operating programs. Automatic segmentation and folding of programs has been shown to be competitive with programmer planned overlays [S1]. The principle of locality briefly states that in general program

references within an interval of time tend to be within confined regions rather than scattered randomly over the name space. As a result, virtual memory systems are empirically justifiable and economically feasible since they offer the memory capacity of a bulk storage device and access rates approximating that of the main memory.

The dynamic storage-allocation scheme associated with virtual-memory systems resolves the name space of the program with the address space at execution time, as segments are brought into main storage. Consequently, a program can be executed even if it is not entirely contained in main memory or if its segments do not reside contiguously in main memory.

1.5.1 Addressing

The mechanism used to map virtual addresses onto main memory addresses is basically the same in all virtual memory systems. Each user in the system has an ordered segment table containing an entry for each of his segments. An entry is empty if the associated block is not in main memory. Otherwise, the entry contains the address in main storage where the block is located.

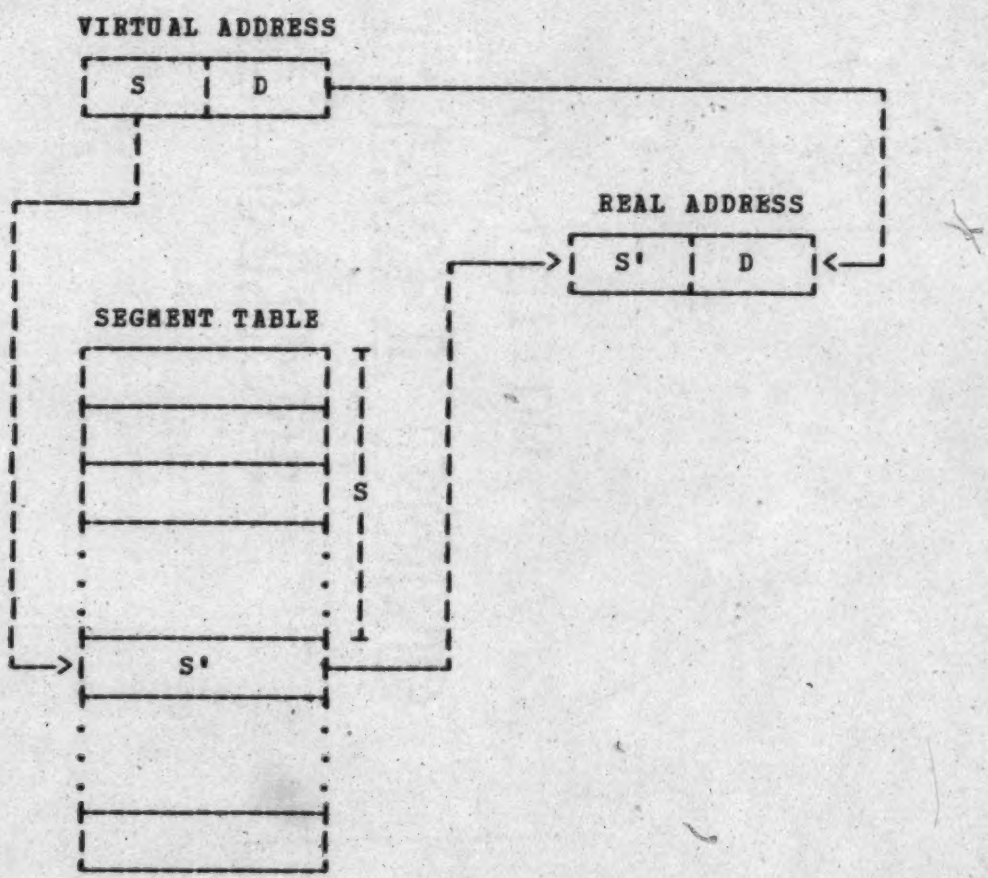
All virtual addresses are two dimensional since they specify a segment and a displacement within the segment. The association of a main memory address to a virtual address is performed via the segment table as shown in

Figure 1. The segment number is used as an index into the table. If the block is in main memory then the address in the table entry plus the displacement in the virtual address determines the address of the desired information. A blank table entry forces the system to bring the required segment into main memory and then update the segment table.

The table look-up technique can be costly in terms of the time spent performing the indirect addressing. As a result many systems have incorporated a small associative memory into the address translation mechanism. These high-speed, content-addressable registers are used to hold a list of the most recently accessed segments and their main memory location. Before initiating the table look-up, the virtual address is compared with the entries in the associative memory. If a match on the segment number is found then a main memory address is immediately available without requiring an access to the segment table.

1.5.2 Implementations

Virtual memory systems are usually said to employ a segmentation scheme, a paging scheme, or a combination of both. Segmentation refers to a system in which the segments are of arbitrary length. Paging, on the other hand, makes use of fixed length blocks called pages.



S - Segment number

S' - Real address of segment S

D - Displacement

Figure 1: Basic Memory Mapping Scheme

1.5.2.1 Segmentation

A few modifications to the basic virtual-memory addressing mechanism are necessary for the implementation of a segmentation scheme. In addition to the main-memory address, the segment size and bits to specify accessibility will be contained in the segment table entry. The displacement in the virtual address can then be checked against the segment length, ensuring that no access will be made outside the bounds of the segment.

One of the problems encountered with a segmentation scheme is the potential inefficient usage of main memory. At any period in time, memory contains user segments and empty blocks or holes. A hole is the space which remains when a segment overlays a larger segment or hole. Individually the holes may not be significant but collectively they may typically consume up to 10% of the main memory capacity [K3]. In order to combat this external fragmentation [D1, R2], memory may be compacted periodically to coalesce the holes into one.

Segmentation schemes were implemented on the Rice University Computer [I1, I2], the Burroughs B5000 and carried on to the B5500 and B6500 systems. In the Burroughs machines, programs are segmented into logically distinct program and data elements by the compilers. The segments are variable in length up to a maximum of 1024 words.

Each job in the system has a table of its addressable segments as its first segment. When a job receives control of the CPU the base address of this segment is placed in a special register. Thus, a program's segments are protected from unauthorized access by other programs. Memory compaction is not employed in the Burroughs scheme, although adjacent holes are merged.

1.5.2.2 Paging

In a paging system the equal-sized blocks or pages in the virtual-address space are transferred to and from blocks of main memory known as page frames. The addressing scheme is identical to the basic virtual-memory addressing model shown in Figure 1. The page table entries may contain extra bits specifying whether a page may be read, overwritten or both. Associative registers can be as valuable in paging systems as in segmentation systems to reduce costly page table references.

Paging systems do not suffer from the problem of external fragmentation. However, a situation known as internal fragmentation is the result of programs being forced into fixed-length pages [D1, R1]. Fragmentation of this type can occur when a program does not exactly fill an integral number of pages. The empty space at the end of the last page may be wasted. Even in systems where pages are packed with program subroutines, procedure blocks or data

areas, each page can contain unused space since the subroutines placed in the pages will not conform to the length of the pages. In the MTS system implemented on the IBM 360/67 the unused space within pages is recorded in an available space list. If the user makes dynamic requests for space, the system tries to satisfy the request via the available space list before allocating new pages. Many studies concerned with reducing internal fragmentation to a tolerable level have been conducted to determine the optimal page size [D1, H1].

Implementations of paging systems have occurred as early as 1961 with the University of Manchester's Atlas computer. The one-level storage system [F2, K1, K2] implemented on that machine has clearly played an important role in the development of virtual memory systems. The Atlas memory system consisted of a 16K core store linked with a 96K drum memory through a paging mechanism. Since the machine was not multiprogrammed, the provision of a virtual memory was merely to increase the address space available to a program. The mapping mechanism employed 32 registers each of which contained the number of the page within the page frame associated with that register.

Another implementation of a paging scheme was on the experimental IBM M44/44X [B4, R1, O1]. In order to study the effects of varying the page size, the system allowed for the specification of page size at initiation. Also, special

instructions were included to indicate the upcoming need for a particular page or that a certain page would no longer be required in main memory.

Control Data's Star-100 and Star-1B [C4] also employ a paged virtual memory system. The Star-1B is a scaled-down version of the Star-100 but its memory-mapping mechanism is practically the same. The pages in the Star-1B contain either 512 or 8192 64-bit words. The page size is selectable under program control. All pages currently allotted space in central storage are assigned "associative words" in the page table. These associative words contain the real core address, page number and user identification and protection bits. The page table consists of 4 associative registers (16 in the Star-100) plus a table located in main memory. If the virtual address cannot be resolved within the associative registers, the entries in the central storage table are "streamed" through the associative compare hardware, which is comparable to a single-register associative memory. The technology of the CDC machine allows table searching to proceed at a rate of two entries every 40 nanoseconds.

1.5.2.3 Segmentation and Paging

It is possible to combine the advantages of both segmentation and paging by combining the schemes within an implementation of virtual memory [A3, D1]. This addressing scheme is shown in Figure 2. The virtual address has three components: a displacement, a particular page, and a specific segment. The segment number is an index into the segment table to retrieve the address of the required page table. A presence bit in the segment table entry indicates whether the page table is in main memory or if it must be fetched from auxiliary storage. The page number in the virtual address is an index into the page table to obtain the address of the required page. Again a presence bit signals the presence or absence of the page in main memory. Finally, the displacement and the page address are used to point to the desired information.

This addressing scheme could potentially cause three memory accesses in order to resolve a single reference. The feasibility of the system depends on the use of associative registers to hold a list of the most recently used pages; in particular the current page from which sequential instruction fetches are occurring.

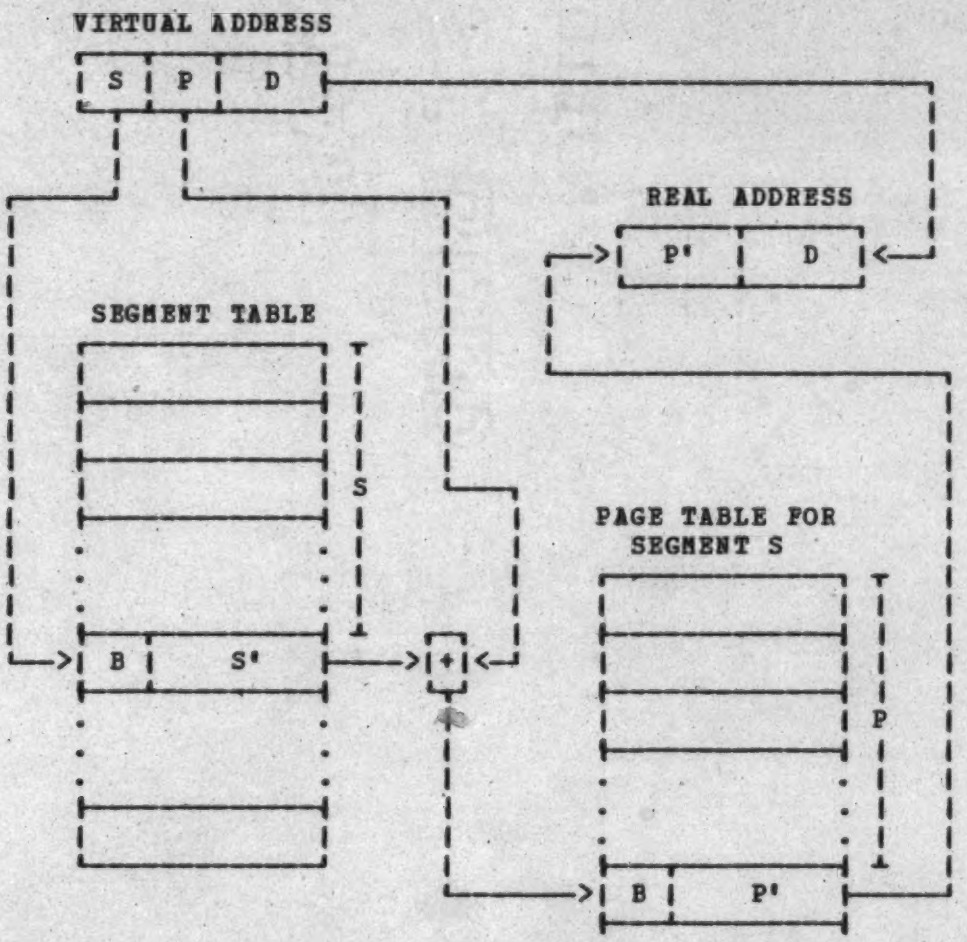


Figure 2: Memory Mapping for Segmentation and Paging

Particular implementations of this type of virtual memory system include the Multics system on the GE 645 [O2, R1] and the MTS or CP-67 systems on the IBM 360/67 [A2, I5, P2, R1]. The Multics system employs dynamic segments ranging to a maximum size of 256K words. The unit of allocation is the page, which was originally either 64 or 1024 words long. More recently a single page size of 1024 words has been used. The purpose of two page sizes was to cut down on internal fragmentation. The IBM 360/67 also employs a 1024-word page and offers 256-page segments. Under 24-bit addressing a total of 16 segments is available (4096 segments with 32-bit addressing). Segmentation in the 360/67 is not based on the structural characteristics of the program or data, since independent programs can be placed in the same segment. The 360/67 employs an array of 9 associative registers, one of which is the instruction counter. The instruction counter contains a relocated address which allows the fetching of instructions without incurring the overhead of address translations. However, a branch within the currently executing page, or an instruction fetch from another page costs an additional 150 nanoseconds if the page address is recorded in an associative register. A reference which is not resolved in the associative registers causes accesses to the segment and page tables plus 750 nanoseconds spent in calculating the appropriate table entry addresses. If the requested page is in main memory an additional memory cycle is required to

fetch the information; otherwise, execution must be suspended until the page has been brought into memory.

1.5.3 Additional Considerations

Modern virtual-memory systems have reduced the need for programmer concern of storage management. From the systems viewpoint, machine resources are allocated only as they are needed. In return, potential resource usage has increased since the multiprogramming capabilities have been improved.

Various problems have arisen in conjunction with the attempt to make the best possible use of hardware resources. These problems are related to the algorithms used to decide which page will enter main memory next, where the page entering memory will be placed and which page will be removed from main memory to provide space for the incoming page. Current investigations in this field have also resulted in the suggestion of bringing pages into main memory that will be required shortly but have not yet been requested. Relatively little attention has been paid to the optimization of programs which operate in a virtual memory system. Virtual memories, especially paging systems, have introduced new constraints on program optimality. Programs which ran efficiently on earlier systems may now incur extensive system overhead in a paging machine.

CHAPTER 2

PAGING

2.1 Introduction

Paging systems, as well as other virtual-memory implementations, have a certain amount of overhead associated with them. The primary problem in implementing an effective and feasible paging scheme is the minimization of overhead brought on by address translation, internal fragmentation and page swapping.

Address translation mechanisms have been improved with the addition of hardware in the form of associative registers to speed up the mapping of virtual addresses onto main-memory locations.

The problem of fragmentation is one which must be delicately balanced with the cost of swapping pages between main and auxiliary memory. As the page size is reduced, internal fragmentation is also reduced since the program segments are more likely to occupy entire pages. Also, with a smaller page size, a page fetch is less likely to result in as much information that will not be referenced being brought into main memory. However, a program will now consist of many more pages and, in addition to the increase in page table size, considerably more time may be spent transferring pages into main memory. Hatfield [H1] concludes that the memory referencing pattern of a program

is also influential in determining the optimal page size.

Finally, the area which has received the most attention recently is the management and movement of pages. When an address translation fails because the required page is not in main memory, a page fault occurs. At this point, the page management algorithm must locate the desired page in auxiliary memory and bring it into main storage. In the event that there are no empty page frames, a decision must be made regarding which page should be removed from main memory. Such a decision is important since it is not desirable to remove a page which will be required in the near future.

Thus, it can be said that the effectiveness of a particular paging scheme depends not only on the speed of address translation but on proper memory management as well. Efficient memory management implies keeping page frames occupied with active code and data while minimizing the number of page fetches.

2.2 Paging Policies

The algorithms which together perform memory management in a paging system are collectively referred to as the paging policy [A1]. The paging policy is composed of the following three parts: fetch policy, placement policy and replacement policy.

Fetch Policy

The fetch policy of a paging system determines when a page is to be brought into main memory. Two classes of this policy are "demand paging" and "prepaging". In most cases, the page fetch is a result of a page fault: this scheme is known as a demand paging policy. In order to implement prepaging, in which pages are fetched prior to their actual need, the system must have information regarding the program's reference pattern. This a priori knowledge cannot include information which is data-dependent, so these schemes do not usually have significant predictive power [D2].

Placement Policy

The placement policy determines which available page frame will hold an incoming page. The appropriate tables must also be updated to reflect the actions performed. The lack of a free page frame temporarily stalls the action of the placement algorithm until the replacement algorithm has been executed.

Replacement Policy

The most important part of the paging policy determines which pages will be removed from their page frames to make main storage available to other pages. Various algorithms have been considered including random replacement (RAND),

replacement of the least recently used page (LRU), and replacement of the page longest in main memory (FIFO, first-in/first-out). The optimal page replacement policy (MIN), proposed by Belady [B1] relies on page-reference information in order to replace the page which will be referenced farthest in the future. Denning's working-set algorithm [D2] would remove a page which had not been referenced within a certain period of time.

The goal of these increasingly sophisticated policies is to ensure that pages which will be required again shortly are not removed from their page frames. As a result, the CPU spends less time awaiting the completion of I/O activity caused by a page fault and is available for problem program processing.

Another method of reducing the amount of page traffic is to maintain a record of all page alterations. If a page, selected to leave its page frame, has not been altered during its period of residence in main memory, then it need not be written into the auxiliary memory. As long as there is an exact copy of the page in the auxiliary storage, the page frame can be overlayed by the incoming page.

2.3 Page Replacement Algorithms

Since the Atlas implementation, the majority of paging systems have employed a demand-fetch policy. The prepaging concept, which is characterized by the loading of pages into main memory prior to a request for them, has not met with much success. The effectiveness of this scheme is limited by lack of knowledge of future page reference activity. Oppenheimer and Weizer [03] discuss the implementation of a prepaging algorithm on the RCA Spectra 70/46 TSOS; however, the 3-msec-per-page loss encountered for each incorrect page fetch postponed use of more extensive prepaging. The placement policy is merely a matter of placing an incoming page into an available page frame. As a result, the replacement policy has emerged as the most crucial component in deciding the effectiveness of the overall paging scheme. It is usually the replacement policy which is examined when the performance of the overall paging policy is questionable.

The basic goal of a memory-management scheme is to keep main memory occupied with program code which can be executed by the processor. Unfortunately, in attempting to satisfy this goal, the paging policy can create many problems for the system.

One such problem is the inefficient usage of working memory. If the pages in use contain only a small percentage

of information actually being referenced then the main memory is certainly not being used to full advantage. Since program code is not executed serially, but rather, consists of regions which are used often, such as in loops, and regions which may never be used, as error-handling routines, pages may suffer the consequences of poor page-loading algorithms that allow high-use code to be placed in the same page as low-use code. Known as dynamic fragmentation, this particular problem can be as harmful as internal fragmentation but is much more difficult to identify and control.

Another resource which is subject to misuse is the central processor. A situation known as "thrashing" [D1, D3] may occur when too many programs try to keep too many of their pages in working storage at the same time: one program's request for pages may result in pages vital to the operation of another program being removed from main memory. The problem of thrashing is one of main memory overcommittment which eventually leads to total system degradation as the processor must continually spend time moving pages between main memory and auxiliary storage.

Denning claims that the reasons for thrashing are a lack of main-memory capacity and the low access rates of the auxiliary storage devices as compared to that of main memory [D3]. Another possible reason is that the programs unfortunately reference a great many pages within a short

span of time because closely related code and data are spread across several pages.

Many studies have concerned themselves with the proposal and evaluation of page replacement algorithms [A1, B1, M1]. Most of the evaluation studies examine RAND, FIFO, and LRU. The Multics paging algorithm, MIN and the working set algorithm are more exotic replacement policies which attempt to overcome the inefficiencies of the former schemes. It is usually the case that the better a particular replacement algorithm is, the more difficult it is to implement.

2.3.1 RAND - Random Replacement

The basic assumption of the random replacement algorithm is that a program's references are uniformly distributed over its entire set of pages. However, it has been shown that program reference patterns are definitely not random in nature but tend to be localized for certain time periods [D2]. The random replacement policy makes its decision based upon an arbitrary rule rather than considering the program's paging history and is therefore classed as being static. Another example of a static replacement algorithm is one which cycles through the list of page frames removing the next page on the list when necessary.

2.3.2 FIFO - First-in/First-out Replacement

Another static replacement algorithm is the FIFO scheme, which replaces the page that has spent the longest time in main memory. This method assumes that programs execute in a sequential fashion and that the page which has been in main memory the longest will least likely be useful. However, the assumption does not accurately model program behaviour. In fact, references tend to be scattered over several pages and a page may be referenced continually throughout its residence in main memory [C1, F1].

A replacement algorithm (BIFO, Biased First-in/First-out) tested on the M44/44X [B2, B4] introduces a bias into the FIFO scheme. A particular program is arbitrarily accorded special status for a certain interval of time. During this interval the FIFO replacement algorithm does not consider any pages of that program for removal. Early results indicated that somewhat better throughput was attained under the BIFO replacement policy.

2.3.3 LRU - Least Recently Used Replacement

The static algorithms RAND, FIFO and BIFO are very easy to implement; however, since they make no use of page reference history, their efficiency is low. The LRU algorithm naturally contains added complexity and overhead since a record of page usage must be maintained and utilized. Because of the amount of overhead, actual

implementations have merely been approximations to the LRU algorithm.

An example of such an approximation is the replacement scheme employed by Multics. Corbato [C3] describes a class of algorithms which employ a k-bit shift register for every real page. Associated with each page is a usage bit which is set to one by the processor hardware whenever the page is referenced and reset to zero when the page is considered for removal by the replacement algorithm. When the replacement algorithm considers a page for removal, it shifts the contents of the related shift register one bit position lower, then deposits the usage bit in the high order position. Only if the shift register is zero will the page be removed from working storage. This particular algorithm reduces to a FIFO scheme if k is zero and approaches the LRU algorithm as k approaches infinity. The Multics system operates with k equal to one.

Another approximation to the LRU algorithm is used by the IBM 360/67 in placing and replacing information in the associative registers [I5]. Each of the eight registers contains a validity bit and a recent-usage bit. An associative register is considered to hold valid information only when the validity bit is set, which occurs when the register is loaded. Since all validity and usage bits are automatically reset when the processor switches from one job to another the possibility of a program accessing a previous

user's page is eliminated. The recent-usage bit is set when the register is loaded and on any subsequent reference to the page addressed via that associative register. When all registers contain addresses to active pages and a new page is referenced, its address is placed in an associative register whose usage bit is unset. Note that the case in which all usage bits are set is avoided by resetting all usage bits whenever the setting of any bit would result in them all being set. The algorithm maintains the most recently used page addresses in the associative array and in some sense removes the least recently used page addresses.

2.3.4 Working Set Algorithm

The problem of reducing page traffic has been regarded as the problem of removing only those pages which will be required farthest in the future. The LRU and FIFO algorithms make basic assumptions about program behaviour in order to estimate which pages will not be needed in the immediate future.

Denning's working-set algorithm employs the principle of locality. Programs execute within certain regions fetching instructions and data from those areas for significantly longer intervals of time than for the transitions between the regions. Also, programs exhibit a tendency to loop through certain regions and sets of pages [D1].

The working set of a process at time t is defined as the collection of information referenced during the time interval (t-t',t) [D2]. In terms of paging, the working set would be the collection of pages referenced within the specified time interval t'. As t' is reduced, the number of pages referenced should also decrease. Under the assumption that immediate past page-reference behaviour is a good indicator of page-reference behaviour in the near future, then as t' is reduced, the predictive power should be increased. Obviously if t' is taken as the length of time to execute the entire program then the working set will be all of the program's pages referenced during the run.

Denning claims that knowledge of the working-set size is sufficient to ensure good memory management [D2]. The program is not initiated unless there are enough page frames available to hold its working set. Knowledge of the working-set size allows the memory-management scheme to reserve enough storage for the program's working set without disturbing the working set of other programs.

2.3.5 Optimal Replacement

The optimal replacement algorithm, when faced with the situation requiring the removal of a page from main memory, chooses one which is no longer needed or will not be needed for the longest time. Therefore, a complete knowledge of future page references is essential, but this is not

available prior to the program's execution and may also fluctuate from one run to another. The previously mentioned algorithms try to guess what the future reference pattern will be, according to fixed rules or estimates based on past behaviour.

The optimal algorithms MIN, proposed by Belady [B1], and OPT, described by Mattson et al. [M1], are not capable of being implemented on a real computer system because the information they require is not available. They are proposed in order to serve as a standard against which the efficiency of realizable algorithms can be compared.

Belady concludes that a good replacement algorithm is one which is a compromise between the simplicity of the random-replacement algorithm and the complexity of an algorithm accumulating page-reference data. The conventional algorithms which are simulated in Belady's study generate two to three times as many replacements as the theoretical minimum, generated by MIN.

2.3.6 Adaptive Replacement

Recent investigations have been in the area of adaptive page-replacement algorithms. Thorington and Irwin [T1, T2] describe an algorithm called SIM which actually contains four replacement policies, LRU, LFU (least frequently used), MRU (most recently used), and MFU (most frequently used), only one of which is the "active" policy. The active policy

controls page replacement in real main memory while the others operate on three simulated main memories.

Each simulated memory is actually a record of the pages that would have been in real main storage had the associated replacement algorithm been in effect. The contents of the real and imaginary memories are monitored by four status bits attached to each mapping-table entry. One of these is used to indicate the presence or absence of that page in memory. Also, each algorithm has a counter which records the number of page faults encountered as it oversees its associated memory.

Typically, the pages present in each of the memories will be different since each algorithm has been removing the page which it considers to be the best choice. The fault counters are constantly checked to see whether one of the inactive policies is performing significantly better than the active one. When a new active policy is initiated, the contents of the simulated memories are updated to be the same as the contents of the real main memory, and the fault counters are set to zero.

Simulation experiments with SIM showed that it outperformed LRU and compared favourably with the best non-lookahead algorithms [T1, T2].

2.4 Conclusion

The attempt to reduce paging traffic has been focused primarily on the replacement algorithm. Although newer algorithms have come closer to the theoretical optimum, improvements have been small, so attention has turned to the areas of program behaviour, structure and locality of reference.

CHAPTER 3

PROGRAM BEHAVIOUR IN A PAGING ENVIRONMENT

3.1 Introduction

Up to this point the discussion has centered on the implementation of an operating system capable of handling a wide variety of jobs efficiently. The intent has also been to develop a system which eliminates the need for the user to know exactly how his program is being processed. Paging systems were thought to be the satisfactory product of these intentions.

However, the performance of a system may be subject to a marked degradation when executing programs which disregard the paging mechanism. The determined effort to improve page replacement algorithms revealed the underlying importance of program behaviour on the overall system performance.

Program behaviour, which can be characterized by the sequence of references made to the program's address space, has been examined from several viewpoints. The study of replacement algorithms considered references made from one page to another as being sufficient to model the program's behaviour [V2]. Other approaches deal with the individual instructions or modules of the program.

It is the contention of this thesis that program behaviour can best be examined through a study of the

interaction between program modules. These modules can be exemplified by FORTRAN subroutines, ALGOL procedures or ASSEMBLER control sections and will be referred to as "sectors". The reordering of these relocatable entities has been demonstrated to affect the program's locality of reference and consequently the program's behaviour [C5, H2]. If program references are made within a confined locality, such as a page, for a period of time then there is less likelihood of causing as many page exceptions.

Thus, if program behaviour can be determined by the knowledge of how the modules interact, then an improved pagination can be proposed. The term pagination refers to the assignment of code and data to pages within the virtual memory. The optimal pagination has the characteristic that the minimal number of transfers between pages will occur.

3.2 Representation of Program Structure

The structure and behaviour of programs has often been represented by directed graphs [R3, K5, V3]. The program sectors are represented by nodes in the graph. The transfers or references between sectors are described by the arcs between the nodes. An illustration of this representation is shown in Figure 3.

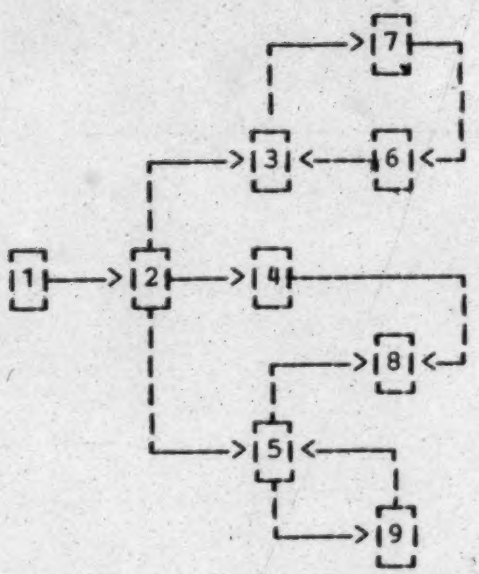


Figure 3: Graphical Representation of Program Structure

The program graph can itself be represented by a connectivity matrix. The matrix elements can be one of two types depending on whether a Boolean or a probabilistic program model is under consideration.

The Boolean model, as examined by Ramamoorthy [R3], Ver Hoef [V3] and Lowe [L1, L2] assigns a 1 to element c(i,j) of the connectivity matrix C if there is a directed arc from node i to j. This represents a nonzero probability that sector i will transfer control to (or reference a data item in) sector j. The program graph in Figure 3 is shown in Boolean connectivity form in Figure 4. The references from node i to node i have not been considered; thus, the elements on the main diagonal of the matrices are represented by zeroes.

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	0	0	1	1	1	0	0	0	0
3	0	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	1	0
5	0	0	0	0	0	0	0	1	1
6	0	0	1	0	0	0	0	0	0
7	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0	0	0

Figure 4: Boolean Connectivity Matrix

	1	2	3	4	5	6	7	8	9
1	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.3	0.5	0.2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.8
6	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0

Figure 5: Transition Probability Matrix

An extension of the Boolean model assigns a weight to each of the directed arcs [R3, K5]. The weights represent the probability of a branch or reference from the sector being executed to another sector. This model depicts a program as a finite Markov chain. As a result, the connectivity matrix is modified to take the transition probabilities into account. The matrix element c(i,j) now corresponds to the arc weight from node i to j. By assigning weights to the arcs in Figure 3 the transition probability matrix in Figure 5 could be derived.

Ramamoorthy [R3] determines sets of nodes which are strongly connected (ie. there exists a path from each node to every other node in the set) and which would best be brought into main memory at the same time. The technique actually being explored was that of fetching the next set of strongly connected sectors prior to a request being issued.

Ver Hoef [V3] presents an algorithm for allocating sectors to pages while minimizing the number of possible inter-page references. The model employed is strictly Boolean; however, the fitting of variable-sized sectors into pages of a fixed size imposes an additional constraint.

Kernighan [K5] employs a directed graph with probabilities assigned to the arcs in order to develop an algorithm to optimally partition the graph. The nodes of the graph have values attached corresponding to the size of

An extension of the Boolean model assigns a weight to each of the directed arcs [R3, K5]. The weights represent the probability of a branch or reference from the sector being executed to another sector. This model depicts a program as a finite Markov chain. As a result, the connectivity matrix is modified to take the transition probabilities into account. The matrix element c(i,j) now corresponds to the arc weight from node i to j. By assigning weights to the arcs in Figure 3 the transition probability matrix in Figure 5 could be derived.

Ramamoorthy [R3] determines sets of nodes which are strongly connected (ie. there exists a path from each node to every other node in the set) and which would best be brought into main memory at the same time. The technique actually being explored was that of fetching the next set of strongly connected sectors prior to a request being issued.

Ver Hoef [V3] presents an algorithm for allocating sectors to pages while minimizing the number of possible inter-page references. The model employed is strictly Boolean; however, the fitting of variable-sized sectors into pages of a fixed size imposes an additional constraint.

Kernighan [K5] employs a directed graph with probabilities assigned to the arcs in order to develop an algorithm to optimally partition the graph. The nodes of the graph have values attached corresponding to the size of

the associated sector. The problem is to partition this directed graph such that the number of transitions across the partitions is minimized and the sum of the sector sizes within the partition does not exceed a fixed value. Since the graph is assumed to be ordered, the task is merely to insert the page boundaries.

Hatfield and Gerald [H2] employ a "nearness" matrix, which is analogous to the matrix of transition probabilities. The entries are counts of actual inter-sector references obtained during a particular execution of the program. A change in the order of the sectors causes the nearness matrix to be modified through a reordering of its rows and columns. Corresponding to the load ordering that groups together sectors which interact often is a nearness matrix whose large entries cluster about the main diagonal. The approach used in the study was to minimize the number of inter-page references by packing the pages with the sectors which reference each other often.

3.3 Gauging Program Behaviour

The initial problem with any of the previously mentioned techniques for reducing page exceptions is that of obtaining sufficient information to characterize the program's behaviour. Kuehner and Randell [K6] suggest possible classes of information that may be useful in determining a program's reference activity.

These include:

(1) frequency of references to sectors

(2) sequence of references to sectors

(3) frequency of references to pages

(4) sequence of references to pages

The latter two are not at a level low enough to disclose the behaviour of sectors within a page. What is really desired is the frequency of references from each sector to each other.

There have been several suggestions on how to obtain the connectivity matrices. First, the programmer could make an educated guess about his program's behaviour. Second, the compiler could gather data about possible inter-sector references. Finally, the most comprehensive technique involves monitoring the program during the course of its execution.

The first two methods are able to provide the information necessary to construct only the Boolean connectivity matrix. Programmer estimation of the number of inter-sector references may be unreliable. On the other hand, the compiler cannot predict reference frequencies or branching probabilities due to inherent data dependencies.

Even execution-time collection of program-behaviour statistics is subject to the criticism that data-dependent structures prevent any generalizations about sector

interaction. The only way to counter such a criticism is to demonstrate the effectiveness of the reorganization of program sectors over a range of data.

Three studies have shown that the reorganization of a program's page contents on the basis of run-time statistics can indeed be beneficial. The experiments, described by Comeau [C5], Peters [P3], and Hatfield and Gerald [H2], are significantly different in the way program-behaviour data was collected.

The first attempt at rearranging page contents, in order to cluster sectors which reference each other frequently, was performed by Comeau [C5]. As such, it was a primitive approach to the problem. The statistics gathered on the Cambridge Monitor System included a history of page requests, page removals, instruction-counter contents at the time of each page demand, and a list of all pages in main memory immediately following a page demand. The memory map also indicated which pages had been referenced since the previous page demand.

The experiment involved assembling and running a program whose sectors were arranged according to four different policies. The run in which the programmer decided the load ordering resulted in fewer page faults than either random or alphabetic ordering. However, when presented with the collected paging data, the programmer was able to

reorder the subroutines within three runs, resulting in an additional 50% reduction in the number of page transfers.

The experimental system described by Peters [P3] for collecting and analyzing data from actual program execution consists of several components. A modified compiler dissects the test program into sectors and determines the Boolean connectivity matrix by inspecting each instruction and noting whether or not it can result in a branch to another instruction sector or a reference to a data sector. At these points in the code a call to a data collection routine is inserted. Thus, when the program is run, a comprehensive history of inter-sector referencing is produced. The reduction of this execution history into a transition matrix is the responsibility of another set of programs. Finally, a set of algorithms which vary in their degree of sophistication perform page packing, utilizing the data gathered.

Unfortunately, although very thorough, the monitoring approach is very costly to the point of being completely infeasible. This is due to the overhead incurred at each inter-sector reference or branch. A large amount of CPU time must also be spent analyzing the data collected. Therefore, it was concluded that the method employed is not suited as a practical device for program reorganization but nevertheless served its purpose as an investigative tool.

The approach employed by Hatfield and Gerald [H2] to collect branching and reference data requires a full trace of the program. This tco is a very expensive procedure, because of the interpretive execution of the program. Similarly, the reduction of the trace into a "nearness" matrix is a computationally expensive task. An additional refinement in their approach is the use of a graphical display of code usage. After preliminary reorganization of the page contents, the program's use of storage can be monitored through an on-line display and additional modifications made to improve the reference behaviour.

In a study of computer performance evaluation, Winder [W1] describes a trace program developed to monitor each instruction execution and address reference by some subject program. The expense involved in running such a trace was balanced by the number of applications in which the data could be employed. The library of address-trace tapes was used to drive "cache system" simulators, to aid in paging studies, to evaluate various processors and also to provide program statistics such as instruction frequencies, program lengths, buffer sizes, successful branches, etc.

The research spawned by the collection of the trace data included a preliminary study of program behaviour via an activity profile and a map of page usage. Although the basic efforts were not directed in the area of program behaviour, Winder acknowledges "a great potential for using

these tools to enhance commonly used programs".

3.4 Monitoring Program Execution

In the previous section, various techniques used in other studies to determine program behaviour were described. The very nature of this research places limitations on the resources which can be used to gauge a program's sector reference pattern. Since the static reorganization which is being investigated can hopefully be performed automatically, it is imperative that both the program monitoring and page reorganization be executed with no external intervention.

On the basis of results from previous studies of program reorganization [K2, P3], it is concluded that in practice the technique cannot be applied to all programs. Therefore, the reorganization is proposed for large heavily used systems such as compilers, loaders, text editors, etc.

The emphasis is placed on the reorganization of object code as opposed to recommending a rewrite of the source language code. Consequently, the only information required to determine a program's reference pattern is its object code and a load map indicating the location of each sector in virtual memory. From within this framework a suitable program monitor to gather experimental data was sought.

Unfortunately, the execution history of a working program is not readily obtainable. The facilities for providing such feedback are just at the stage of being implemented in hardware and software systems.

Ingalls [I7] mentions several software systems which have been built to provide execution-time "profiles". These profiles merely reveal the frequency of execution of individual instructions (or statements in the source language code). The same system is also described by Knuth [K4]. Although the idea is simple, and apparently easily implementable, it does not provide adequate information for the purposes of this study. A record of instruction or sector usage cannot replace an accurate record of the references between sectors.

The monitor system of Peters [P3], described in the previous section, was also rejected since it is forced to manipulate the source code. Such a system is clumsy and wasteful, because massive amounts of extraneous code must be added to the original program.

Execution monitoring was eventually performed by a software system similar to the one described by Winder [W1]. The generation of a complete address trace provides the ultimate description of the program's reference activity. Winder was in the advantageous position of being able to use the RCA Series 70 computer, which has special

hardware features facilitating the development of a program monitor. It was reported that monitoring of a program required 20 to 30 times as much execution time as a run of the program itself. In contrast, another monitor written for the timesharing system (TSOS) on the RCA computer could require 200 times as much execution time for compute-bound jobs.

For experimentation purposes, software monitors have been considered; however, for practical purposes some form of hardware or microprogrammed monitor might be better suited to the task. Jasik [J1] discusses a hardware monitor for the CDC 6000 computer which gathers a profile of a program's execution. The basis of the monitor is a peripheral processor which is programmed to read and record the program address register of the CPU program being observed. The report reveals an underlying capability of a hardware monitor to gather statistics on a program's execution without actually affecting the job. Reports on the Burroughs B1700 [W2, W3] describe a profile-statistics-gathering mechanism which can be used in any language interpreter on the system. As well, the microprogramming capabilities of the machine may allow dynamic measurement of software "events" specified by the user.

Demand for high performance program monitors will come from two main sources. First, programmers wishing to improve source code will require feedback revealing which

parts of their program are being executed unnecessarily often or have never been tested. Second, those interested in improving the performance of programs in paging systems via a reorganization of the object code (sectors) will also require execution time statistics.

For the purpose of gathering experimental program reference statistics, two software monitors were considered and ultimately one was used. The first one examined produces an interrupt-driven trace of all transfers of control between sectors. Prior to execution of the subject program the monitor system inserts a trap at each location which can result in a branch to another sector. As the program is executed under control of the monitor, each trap is recorded as a transfer between two specified sectors. The obvious problem with this system is that it fails to recognize data references as a cause of inter-sector references. Even though the full significance of external data references was not apparent, this monitor was not considered further. Any modification to markedly improve the capabilities of the interrupt-driven monitor was deemed impractical.

It was decided that the most reliable approach to obtaining the complete sector reference string was from a full address-trace of the program. The monitor implemented controls the program's execution by interpreting each instruction. The software package [M2] was originally used

to tally the usage of individual machine instructions within the subject program; however, with several modifications, a trace of each location referenced within the program's name space was also collected. When the trace is analyzed in conjunction with the load map of the program, a matrix of inter-sector references can be built.

At this point it may be interesting to mention information obtained which is relative to the original concern about the significance of data references. In the following chapter, detailed statistics on instruction types used by monitored programs will be presented; however, at this point brief mention will be made of general program characteristics. As program code is executed, references to main storage occur as a result of either instruction or data fetches. Naturally, if the program can maintain its data in working registers then fewer memory references will be required. The experimental results would lead one to believe that ASSEMBLER-written programs control register usage better than programs of a higher level language such as ALGOL. The reference-to-instruction ratio for ASSEMBLER programs is 1.45:1.0, indicating that on the average the execution of an instruction necessitates one memory cycle for the instruction itself and .45 cycles for data. Therefore data references can have a severe impact on the number of inter-sector references occurring within a program. Any proposed program monitor should gather data-

reference statistics in order to be valid.

Programs run under the monitor experienced a decrease of execution speed on the order of 50 to 100 times. Programs which had a higher proportion of memory-referencing instructions suffered the most, because of extra address calculations by the monitor. Obviously, this type of monitor cannot be expected to be suitable for large, long running programs due to the cost involved. However, despite its shortcomings, the monitor performed its task of obtaining experimental results.

CHAPTER 4

RESTRUCTURING PAGE CONTENTS

4.1 Introduction

After solving the initial problems of collecting program behaviour data and reducing it to an analyzable form, there still remains the task of proposing a reorganization of the page contents. The ease with which the code can be reordered is highly dependent upon the method used to perform the original pagination. When appropriate assumptions are made or restrictions imposed on the loading method, there may yet be a major computational procedure involved in determining the optimal organization of the code.

Comparable to the case of replacement algorithms in which the optimal scheme is computationally intractable, several sub-optimal but implementable techniques have been devised. In the following sections an analytical approach to the problem and several experimental algorithms are explored, in addition to a brief discussion of loading techniques.

4.2 Assignment of Program Code to Pages

There are two basic ways that program code may be assigned to pages in a virtual-memory system. The first loader algorithm simply fills one page at a time,

disregarding any logical or physical divisions in the code. This amounts to dividing the program into N-word segments, where N is the page length. Thus, the only page which can suffer internal fragmentation is the last. The alternative algorithm takes into account that certain sections of code are movable, that is, the operation of the program is independent of the location of these sections of code. The loader's task is to place these modules into pages. A simple loader will probably use a "first-fit" algorithm to determine where the next sector will be placed. In other words, the sector will be located in the first page which has enough available space to accommodate it. This second type of loader, which will be the one considered hereafter, is known as a "sector loader", and attempts to improve paging performance and can be upgraded so that it packs sectors into pages according to a particular policy.

It is of course assumed that all discussions of program object modules refer to code which has not been processed by a linkage editor. Therefore, the object modules will be composed of relocatable sectors of code which have not had their external references resolved.

Use of a linkage editor poses a very interesting problem in paging systems. The obvious advantages are a saving in processing time by the loader and a minimization of the number of pages required by the loaded program. Potential disadvantages arising from the use of a linkage

editor are poor utilization of the dynamic loader facilities offered by paging systems, and poor placement of code within pages. One critical example of bad code placement occurs when a loop within a sector is placed across a page boundary. From the view of this study, the use of a linkage editor would hinder the attempts to reduce paging traffic caused by inter-sector references.

4.2.1 Pagination to Reduce Internal Fragmentation

A scheme could be devised whereby the sector loader, rather than simply performing a "first-fit" allocation, performs a page assignment which minimizes internal fragmentation and thus minimizes the total number of pages required to hold the program. Since programs in paging systems tend to occupy a minimum number of page frames during execution, this may be a worthwhile type of optimization for small or medium-sized programs. Hopefully, the reduction in the number of pages occupied by such a program would allow the majority of its pages to remain in main memory.

4.2.2 Pagination to Reduce Inter-Page Referencing

Page-packing schemes designed with the goal of reducing inter-page references have been attacked as being attempts to tune the program to the operating system, rather than tuning the operating system itself [K6]. In most cases though, the selective placement of program sectors within

pages in order to reduce the number of page exceptions is intended for large, heavily used programs which indeed might be considered part of the operating system.

If it is assumed, as Kernighan did [K5], that the order of program sectors must not be changed, the problem becomes that of partitioning the program. The partitioning is done such that inter-partition references are minimized subject to the constraint that a partition length does not exceed the fixed page size.

Kernighan shows that the algorithm for computing the optimal partitioning requires execution time which in most cases grows linearly with the number of sectors under consideration. However, this algorithm is likely to cause an increase in internal fragmentation since the page packing may be very loose. The constraint of not allowing sectors to be reordered is particularly undesirable when the sectors are indeed movable. It is this constraint which reduces the computational complexity involved in performing a total reorganization of sector allocation within pages.

If program segments can be reordered, then one can consider collecting sectors which communicate frequently into a common page. As mentioned previously, the component which decides how the sectors will be clustered is the Boolean-connectivity or transition-probability matrix in conjunction with an organizing algorithm. The Boolean

matrix is considerably easier to obtain since most compilers and assemblers produce a list of references which are external to each program sector. While not as easy to obtain, the transition probablilty matrix gives a better idea of how a set of program sectors interact.

Assuming that either a Boolean or probabilistic matrix of inter-sector referencing has been obtained, a major computational problem arises in calculating the optimal assignment of sectors to pages. The problem can be related to the field of discrete optimization and may require an extremely costly integer-programming solution. Several approaches to this problem will be examined in the following sections of this chapter.

4.3 Optimal Sector Placement

The problem is that of reducing or minimizing the total number of inter-page references which will occur during a program's execution by packing pages with the sectors which reference each other the most. Throughout the following discussion it will be assumed that the sector size is always less than the page size. Data sectors may be expected to span several pages on occasion. In fact, if a sector is larger than a page then the sections which must exist in different pages could actually be considered as different sectors.

The problem can be stated in the following way:

Let: $s(j)$ be the size of sector j, $j=1,2,\ldots,N$

$p(i)$ be the set of pages of size P, $i=1,2,\ldots,M$

$T = \{t(i,j)\}$ indicate in which page each sector is placed,

$t(i,j)=1$ if $p(i)$ contains sector j

$t(i,j)=0$ if $p(i)$ does not contain sector j

$B = \{b(k,j)\}$ be the Boolean connectivity matrix,

$b(k,j)=1$ if sector k references or branches to sector j

$b(k,j)=0$ otherwise

$C = \{c(k,j)\}$ be the transition probability matrix such that $c(k,j)$ indicates experimental probabilities of sector k referencing sector j, obtained by monitoring the program

According to the assumption that $s(j) \leq P$, for all j, in the worst case of sector packing, which occurs when only one sector is assigned to each page, M equals N.

Under the following constraints:

1. For each page, the sum of the sizes of the sectors in the page must be less than the page size.
2. Each sector is placed in only one page. ie. no sector duplication

which can be expressed mathematically as:

$$\sum_{j=1}^{N} t(i,j)*s(j) \leq P, \text{ for all } i,$$

$$\sum_{i=1}^{N} t(i,j) = 1, \text{ for all } j,$$

where the symbol "*" indicates multiplication. One possible objective function to be minimized under the Boolean approach is D(B), corresponding to the total number of points in the program which can cause a page exception:

$$D(B) = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} [t(i,j)-t(i,k)]^2*b(j,k)/2.$$

For the transition probability case, the function to be minimized is D(C), which represents the total number of page exceptions that will occur when the program is executed.

$$D(C) = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} [t(i,j)-t(i,k)]^2*c(j,k)/2.$$

The above approach requires a solution for the T matrix which implies solving for N^2 variables. Since each t(i,j) can be either 0 or 1, enumeration of all $2^{**}N^2$ possibilities (** - exponentiation), subject to the stated constraints, is definitely beyond the range of computational practicality for realistic values of N, typically greater than 10.

The number of possibilities can be significantly reduced if the following approach is taken.

Consider the transition probability model.

Since all terms in the summation are non-negative,

$$\text{Min}\{D(C)\} = \text{Min}\{\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}[t(i,j)-t(i,k)]^2 * c(j,k)/2\}$$

$$= \sum_{i=1}^{N}\text{Min}\{\sum_{j=1}^{N}\sum_{k=1}^{N}[t(i,j)-t(i,k)]^2 * c(j,k)/2\}$$

This is equivalent to filling one page at a time while minimizing the number of out-of-page references at each step.

In other words, the second approach solves for T, one row at a time. An upper bound on the number of combinations of zeros and ones for the t(i,j)'s can be obtained by considering the case where only one sector will be assigned to a page. Solving for t(1,j) will require a possible 2**N enumerations. Since one sector will have then been assigned to page 1, it can be omitted from further consideration. Solving for t(2,j) will require a possible 2**(N-1) enumerations. Again another sector can be disregarded. Proceeding in this manner, the total number of enumerations would be

$$2^{**}N+2^{**}(N-1)+\ldots+2^1+2^0 = 2^{**}(N+1)-1.$$

The "Backtrack Half Interval" algorithm (BHI), described by Peters [P3], is another version to produce

optimal sector ordering based on the transition-probability matrix. This approach again involves a search for an integer assignment solution. Using the previous notation, the approach can be described as follows:

Let a(i), i=1,2,...,N be a function such that a(i) is the number of the page containing sector i. The C matrix is assumed to be symmetric; therefore, only those elements c(i,j) where i≤j need be considered. Then D(C), which is a function of the particular packing, is:

$$D(C) = \sum_{i=1}^{N} \sum_{j=1}^{N} c(i,j) * (1 - DEL[a(i), a(j)])$$

where DEL[i,j] is 1 if i=j and 0 otherwise. The object is to minimize D(C) subject to the constraint

$$\sum_{i=1}^{N} s(i) * DEL[a(i), k] \leq P, \text{ for all } k.$$

Since there are N possible pages in which a sector may be placed, each a(i) has N possible values, disregarding the constraints. Therefore, there are N**N possible arrangements which may require evaluation. The BHI algorithm was implemented using backtrack and half-interval techniques described thoroughly by Peters.

4.4 Heuristic Sector Placement Algorithms

Due to the apparent impracticality of solving the sector loading problem by an analytical approach, heuristic methods have been suggested as an alternative [H2, P3]. The heuristic approaches may not be optimal; nevertheless, they afford a potential improvement in paging performance which merits investigation. Of the many heuristic algorithms which can be devised, the one which will most efficiently determine an optimal or near-optimal load ordering is sought.

The "Unit Merge" algorithm [P3] is an example of a method that cannot guarantee an optimal sector loading. The algorithm proceeds as follows:

1. From transition probability matrix C, determine max c(k,j).
2. If s(k)+s(j) > P, then go to 5.
3. s(k) <-- s(k)+s(j)
   c(k,q) <-- c(k,q)+c(j,q), for all q
   c(q,k) <-- c(q,k)+c(q,j), for all q
4. s(j) <-- 0
   c(j,q) <-- 0, for all q
   c(q,j) <-- 0, for all q
5. If for every c(k,j) > 0, s(k)+s(j) > P, then further merges are made solely to reduce the number of pages required; otherwise go to 1.

Experiments conducted with the Boolean, Backtrack Half Interval, and Unit Merge packing algorithms have shown the heuristic approach to be the only practical one [P3]. The Boolean and Backtrack Half Interval techniques each consumed 3 hours of GE-635 CPU time before being terminated without results, while the Unit Merge algorithm required 2 minutes of execution time to determine a load ordering for the same small subject program. While Unit Merge may not provide the best loading scheme, it does increase the number of references within the same page and thus reduces inter-page referencing.

Hatfield's approach is that of reorganizing the C matrix into sub-matrices so that the large elements cluster about the main diagonal, corresponding to pages [H2]. The problem is another which does not lend itself to an efficient procedure for producing the optimal ordering of rows and columns. Once again it is heuristic methods which are actually implemented.

Eigenvalue analysis is suggested as a technique which can be used to identify the sector clusters. The technique has also been used in the field of Information Retrieval to cluster documents or terms [S2]. In each eigenvector of the C matrix there tends to be a group of elements with large values in comparison with the rest. A page is filled with the sectors corresponding to the largest elements of the eigenvector. A figure of merit can be computed for each

assignment by comparing the sum of the eigenvector values for the sectors in the page with the corresponding sum for the sectors outside the page. The assignment with the highest figure of merit is used and the rows and columns of C for the assigned sectors are deleted.

In the research reported by Peters [P3], the Boolean model was also considered. Based on the work of Lowe [L1, L2], Ramamoorthy [R3], and Ver Hoef [V3], the "Boolean Packing Algorithm" first attempts to identify the cyclic structures within the program and then pack the sectors comprising a loop into a common page. Neighbouring loops and the sectors linking them are also assigned to the same page, should their size permit it.

Looping structures are detected by raising the Boolean connectivity matrix to successive powers. In B**N, element b(i,j) is one if there is a path through N arcs between nodes i and j of the directed graph for the program. Also, node i is in a loop of length N if b(i,i) of B**N is one.

Once the loops have been identified, the ones which are less than a page in length can essentially be merged to form a new sector. The next step is to coalesce linearly connected sectors and then consolidate partially filled pages to reduce internal fragmentation.

Obviously, the Boolean model is very closely related to the transition probability model in that the elements which

are one in the connectivity matrix are replaced with probabilities. Intuitively the extra information provided by the probabilistic model should permit an algorithm for better page packing. As Lowe mentions [L2], a comparison of the Boolean and probabilistic methods, including potential performance improvement and complexity of implementation is an area of investigation which deserves attention; however, it is not within the scope of this particular study.

4.5 Practicality of Page Restructuring

An important consideration in developing an automatic sector-packing technique is that of ensuring the practicality of the system. Obviously, the cost of performing the repacking must not exceed the expected savings in the long run. This criterion places a limitation on the type of programs which should be optimized as well as the type of restructuring algorithms which could be implemented.

Analytic solutions to various memory-allocation problems of reasonable magnitude have often been avoided because of their prohibitive cost. Garey, et al. even reject this approach for the relatively simple problem of determining the minimum number of pages required to hold a set of sectors [G1]. Similarly, investigations of the more complex problem of packing pages to minimize the inter-page references have also avoided analytical solutions [H2, P3].

Efforts have been directed at finding a suitable heuristic solution, implying the determination of a set of rules which, when applied to the problem, gives a valid solution that is reasonably optimal. Heuristic methods, when properly designed, are quick and give sub-optimal results within acceptable bounds of the fully optimal. It has been noted that heuristic sector-reordering techniques can reduce the number of page exceptions in the order of two-to-one to ten-to-one [H2]. The improvement is due to the reduced inter-page referencing and more compact working sets. It is noted that the effectiveness of the technique is reduced when the average sector size exceeds half the page size.

CHAPTER 5

ASSESSMENT OF STATIC PROGRAM REOGANIZATION

5.1 Introduction

In this chapter an attempt will be made to establish the effect of a static program reorganization in terms of a reduction in potential paging activity. A study of this nature could in fact centre on any of three areas which influence a program's working-set dynamics and consequently paging activity. These three areas are:

1. Internal fragmentation
2. Density of reference
3. Locality of reference

The argument for minimizing internal fragmentation is a two-fold one. In addition to the obvious objective of minimizing space allocated to the program, it is desirable to maximize the productivity of each page loaded, by fetching potentially useful information rather than regions of wasted space.

Density of reference refers to the amount of allocated storage which is actually used during the course of program execution and could conceivably be measured as the number of unique locations referenced in a page during a specified interval of time. The concern is with regard to how much of the information in the page is really needed. Once again, the purpose of any reorganizatcn based on density of

reference would be to maximize the use of each page loaded by ensuring that the entire page contains needed information.

Locality of reference is associated with the proximity of locations whose contents are referenced close together in time. A program which exhibits a high degree of localization in memory referencing should have considerably less paging activity than a program which scatters its references over many pages.

Since this study is principally concerned with reorganization of a program's pages at the sector or module level, it may show favourable results regarding density and locality of reference. A reordering at this level cannot improve behaviour within the modules since this is a problem associated with design and source language coding. By placing sectors which interact a great deal in the same page, the density of reference as well as the locality of reference for that page may be improved markedly. In stipulating a pagination requiring specific modules to be in the same page, one may observe an increase in the internal fragmentation and consequently total number of pages allocated to the program. However, the important consideration is to reduce real memory requirements and CPU overhead, even at the expense of occupying additional virtual memory.

5.2 Monitor Results

The program monitor which was briefly described in the preceding chapter existed in two forms. Originally the behaviour of the program under observation was recorded as a string of memory addresses tagged as being either the result of an instruction fetch or a data reference. With this monitor the following problems were noted.

1. Costly to run due to large number of I/O requests.
2. Analysis of data could be as expensive as collection of data, because of I/O events.
3. Large files of address-reference strings must be maintained.

The advantage of collecting and maintaining a large data base of program-reference strings is that, subsequently, a great many statistics can be derived.

A decision was made to collect information on the behaviour of several programs in a form similar to the transition probability matrix. Therefore, the analysis of the address reference strings was incorporated within the program monitor. In this way, the overall expense of collection plus analysis was reduced.

While the monitor interprets and simulates the execution of the test program's instructions it fills a

buffer with the memory addresses being referenced. With the use of the load map the analysis stage reduces the string of address references into a matrix of sector interactions. Since the addresses are tagged if they correspond to instruction fetches, it is possible to determine if the occurrence of an address external to the current sector is the result of a data reference or a transfer of control between sectors. While the matrix does not differentiate data references from transfers, a cumulative tally of these statistics is maintained and made available at termination.

The following example illustrates how the monitor constructs the reference matrix and collects additional information regarding branching and referencing between sectors. Consider a program with two sectors, A and B. The following notation will be employed.

a' - instruction fetch from location within A

b' - instruction fetch from location within B

a - data reference to location within A

b - data reference to location within B

The following reference string would result in the generation of the matrix shown in Figure 6.

a'aa'abb'bb'bab'aa'

	A	B
A	4	2
B	3	4

Figure 6: Inter-Sector Reference Matrix

The monitor also provides the following information from the reference string.

Instructions executed: 6

Total references: 13

Intra-sector data references: 3

Sector transfers: 2

Inter-sector data references: 3

Three programs thought to be good candidates for reorganization because of their size and running characteristics were chosen for monitoring. Each program was monitored several times with varying input data. A brief description cf the programs is given below.

Program 1. ASSEMBLER - Information Retrieval Application

Program 2. ASSEMBLER - FORTRAN G Compiler

Program 3. ALGOLW - Chess Playing Program

The various runs of a program are denoted by the program number followed by an alphabetic character, eg. 1a, 1b, 2a. In Table 1 information on the instructions used by the monitored programs is given. Included are the number of instructions executed, number of references made, branching frequency, and usage of register-only instructions.

PROGRAM	#INSTRS	#REFS	REFS/INSTR	%REG	%BRANCHES
1a	22750	33903	1.49	60.7	33.8
1b	53122	80190	1.51	60.4	36.2
2a	154193	223044	1.45	53.5	25.4
2b	435876	627077	1.44	54.1	27.1
2c	478229	687720	1.44	54.0	27.0
2d	579298	833302	1.44	54.2	27.0
2e	1179888	1655916	1.44	54.5	26.7
3a	750461	1198706	1.60	39.4	16.7
3b	759815	1211747	1.59	39.7	17.0
3c	1887033	3009286	1.59	39.7	16.7

#INSTRS - number of instructions executed
#REFS - number of memory references
REFS/INSTR - average number of references per instruction
%REG - percentage of instructions referencing registers only
%BRANCHES - percentage of instructions resulting in transfer of control

Table 1: Program Instruction Types

These statistics illustrate the inherent differences in object code produced for the various programs. The ALGOLW program has a markedly larger reference-to-instruction ratio than the ASSEMBLER programs. Also, the percentage of

instructions which cause branches or which use only registers is much less in the higher-level language program, indicating that data fetches may be more critical than transfers of control in producing inter-sector references. The data gathered and interpreted in Table 2 shows the significance of data references.

PROGRAM	%INT. DATA REFS	%TRANSFERS	%EXT. DATA REFS
1a	10.2	2.2	20.4
1b	12.0	1.0	20.8
2a	4.9	6.7	26.0
2b	5.4	7.3	25.1
2c	5.4	7.3	25.1
2d	5.5	6.9	25.0
2e	5.3	7.3	25.2
3a	2.8	2.7	34.6
3b	3.1	2.6	34.2
3c	2.7	2.8	34.6

%INT. DATA REFS - percentage of references which involve a data fetch from a location within the currently executing sector

%TRANSFERS - percentage of references which involve a transfer from one sector to another

%EXT. DATA REFS - percentage of references which involve a data fetch from a location external to the currently executing sector

Table 2: Program Reference Characteristics

From these examples one can observe that inter-sector referencing including transfers and data fetches may account for as many as 36% of all references. In the worst case one would find that all inter-sector references cause page faults. Of course this can be further complicated by

sectors which cross page boundaries. The only sectors which cross a page boundary are those that are larger than a page.

An example of the reference matrix produced by the monitor is listed in the Appendix. It is the reference matrix for test program 1a.

5.3 Program Reorganization

In order to gain an appreciation for the reduction in potential page faults that one might expect from a program reorganization based on inter-sector reference patterns, the results of the monitored programs were analyzed to obtain several important statistics.

First, observing the loading algorithm for the MTS system on the IBM 360/67, all inter-sector references which could result in a page fault were tallied. Then, for comparison, the page contents were reorganized according to the Unit Merge algorithm described by Peters [P3] and again the possible page faults were calculated. This algorithm was used since it was anticipated that the reorganization would approximate the optimal ordering. The transition matrices were observed to be very sparse and contained sets of elements which are many orders of magnitude greater than the rest. Therefore, the optimal algorithm would possibly afford only minor additional improvement.

In Table 3 below, the total number of inter-sector references as well as the percentages which might cause page faults according to the MTS and Unit Merge loading techniques are presented.

PROGRAM	#EXT. REFS	%FAULTS	%FAULTS*	%IMPROVEMENT
1a	7686	85.0	53.7	36.8
1b	17468	91.5	72.5	20.8
2a	72910	85.7	51.6	39.8
2b	202972	85.5	56.8	33.6
2c	222582	85.0	57.2	32.7
2d	265665	82.2	57.1	30.6
2e	550538	96.8	57.3	40.7
3a	446923	99.6	98.2	1.4
3b	446735	99.8	98.5	1.3
3c	1122673	99.4	97.8	1.6

#EXT. REFS - number of references to locations external to the current sector
%FAULTS - percentage of external references which may cause a page fault because the two sectors are in separate pages
%FAULTS* - identical to %FAULTS except that the pages are considered to be reorganized according to the UNIT MERGE algorithm
%IMPROVEMENT - improvement in terms of reduction in potential page faults afforded by reorganization

Table 3: Potential Page Faults

Several interesting features of the programs which were monitored can be seen in the light of the statistics shown in Table 3. First, the reordering was determined on the basis of the Unit Merge algorithm being applied to one of the transition matrices for a particular program. The various runs of each program used different data in order to

gauge the fluctuation in behaviour. However, the realignment calculated from one run provided a comparable improvement for all runs of the program. Introducing different data to the program did not drastically affect the program's behaviour; though it resulted in new modules being referenced while others were no longer utilized, the interaction between program sectors was not changed significantly.

Program 1 exhibits a higher degree of data dependency than the rest. The results in Table 3 were based on a page loading suggested by the transition matrix for run 1a. If the page contents are specified by the Unit Merge algorithm applied to run 1b then the results are as follows.

PROGRAM	#EXT. REFS	%FAULTS	%FAULTS*	%IMPROVEMENT
1a	7686	85.0	81.4	4.2
1b	17468	91.5	68.9	24.7

Table 4: Potential Page Faults for Program 1 Reorganized According to Run 1b

The FORTRAN G compiler showed a consistant improvement ranging from 30% to 40% and once again variations in the input data caused minor differences in the suggested reorganization. A variety of small FORTRAN programs was compiled in order to trace as many of the compiler modules as possible. Included were several programs with syntactic errors which insured that error routines would be executed.

The ALGOLW program showed practically no improvement although there seemed to be great potential since 99% of the inter-sector references could conceivably cause a page fault. The reason for the lack of improvement was attributed to two characteristics of the program. First, there were several highly used sectors too large to be packed with their associated sectors. Secondly, the dynamic acquisition and release of storage, common to ALGOL programs, results in many references to sectors which do not exist at load time and cannot be reorganized. Further examination of the data for programs 1 and 3 was deemed necessary to confirm the above suspicions.

The concept of a "super-page" was introduced to determine the significance of the large sectors in the overall reorganization. A super-page of size n=2,3,... can be interpreted as either a page of size n*4096 bytes versus the standard 4096 bytes or an association between an n-tuple of pages such that whenever one is in main memory then the others will be too. Reorganization of program 1 was done for super-pages of size 2 and 3 and for program 3 with sizes 2, 3, 4, and 5. The results are shown in Table 5.

PROGRAM	%FAULTS					
	MTS	1	2	3	4	5
1a	85.0	53.7	42.0	35.9		
1b	91.5	72.5	56.4	48.8		
3a	99.6	98.2	97.7	97.5	96.9	94.9
3b	99.8	98.5	97.4	97.2	96.7	94.9
3c	99.4	97.8	97.3	97.2	96.7	94.6

Table 5: Potential Page Faults,
Considering Super Pages

The additional improvement for program 1 provided by the super-page is not as significant as the original reorganizational improvement. In the case of program 3 the only noticeable improvement occurs at super-page size 5, which corresponds to the existence of a very heavily used sector just larger than 4 standard pages. However, even maintaining a minimum set of 5 pages in main storage at all times affords only a 5% improvement.

Several of the programs monitored undertook the management of storage needs by acquiring and freeing memory dynamically (GETMAIN and FREEMAIN commands). These regions cannot be manipulated at load time since they appear and disappear during the course of execution. Nevertheless, the data fetches to these regions can be significant in terms of causing page faults. In order to gain an appreciation for the importance of data fetches to these regions, the monitor data was analyzed once again. Table 6 shows the same type of statistics as Table 3; however, references to regions acquired dynamically by the program have been removed.

PROGRAM	#EXT. REFS	%FAULTS	%FAULTS*	%IMPROVEMENT
1a	7084	83.8	49.8	40.6
1b	15951	90.7	69.8	23.0
3a	49931	96.1	83.7	12.8
3b	49966	98.5	86.9	11.8
3c	130583	94.6	81.1	14.2

Table 6: Potential Page Faults,
Ignoring Dynamically Acquired Regions

While program 1 does not demonstrate a marked improvement when its dynamic regions are not traced, program 3 appears to be limited by its data fetches to program-acquired storage areas. Finally, if the data used to obtain the statistics in Table 6 is analyzed under the super-page concept, then a new set of results is available and is shown in Table 7.

PROGRAM	%FAULTS					
	MTS	1	2	3	4	5
1a	83.8	49.8	37.0	30.4		
1b	90.7	69.8	52.3	43.9		
3a	96.1	83.7	79.4	77.8	72.8	54.2
3b	98.5	86.9	76.6	75.1	70.3	54.1
3c	94.6	81.1	76.8	75.6	71.3	53.2

Table 7: Potential Page Faults,
Ignoring Dynamically Acquired Regions,
Considering Super Pages

In the final analysis it can be said that any reorganizational improvement in the page faulting rate for program 3 is dependent on the size and use characteristics of the sectors as well as dynamic acquisition of memory.

5.4 Interpretation of Program Reorganization Statistics

As the results indicate, it is difficult to generalize on the expected performance improvement due to program reorganization, since individual programs vary greatly in their behaviour and reference patterns. Even in cases where a reasonable 10 to 20 percent reduction in potential page faults is achieved, the actual reduction in number of page faults is subject to variation. If a 50-page program can execute with all its pages in real memory, then no real performance improvement will be evident. At the opposite extreme, if only one page can be maintained in core at a time and the program can execute with a single page frame, then the real improvement is identical to the potential improvement in page fault activity.

An interesting facet of the results on potential paging reduction is the drastic effect that dynamic program storage requests can have on overall improvement. These results would indicate that the ability to acquire and release memory space during execution is a hindrance in accomplishing the goal of this thesis. The original reason for allowing this feature was to limit main memory usage by a program until space is actually needed. However, the paging mechanism has assumed this role. Since virtual-memory occupation charges should be negligible in comparison with real-memory charges, the use of dynamic virtual-memory

allocation might well be replaced with pre-loading specification of all virtual-memory requirements. In doing so, a reorganization of the program would include these previously transient regions and performance improvements in terms of reduction in page faults might be achieved. An undesirable consequence may result: a need for a greater number of page frames at any time.

The best interpretation of the results presented in this study is that there is significant potential for improving the performance of large and often-run systems programs which were originally written to run in other than paging systems.

The determination of the real performance improvement is left to those studies concerned with monitoring actual paging activity. A study of that nature, however, would in fact be involved with validating the principles of locality and density of reference. In this thesis, these principles have been assumed to be correct and thus, the suggested reorganizations should result in more efficient running characteristics [H2].

CHAPTER 6

CONCLUSIONS

Briefly summarizing, it was the intent of this research to explore the idea of improving software performance in paging systems by first determining program behaviour and then reorganizing the logical modules within pages so as to optimize execution. The reorganizational scheme might result in improvement of programs originally written for a different type of operating system. Also, the method is equally applicable to large systems whose modules are programmed by many individuals.

Basically the expected benefit from such a reorganization is a reduction of the number of page faults and program interrupts experienced during execution, which can be interpreted as an improvement in the program's locality of reference. Therefore, once the program receives control of the CPU it should be able to execute for longer periods of time before being interrupted by the need for a page not currently in main memory. In addition, the entire system should experience a gain in total throughput because less overhead in servicing page requests is realized.

Certain programs have a much greater potential for improvement than others. Reduction in potential page faults as high as 40% can be expected in some instances while a meagre 1% improvement may occur for other programs.

The small improvement can be attributed to various characteristics of the programs. The most obvious is that the program already has a very high degree of locality in its referencing pattern, or the sectors of the program may not be of sufficiently small size to allow an effective reorganization. Also, the program may be utilizing techniques which prevent a significant reorganizational improvement, such as controlling their own storage acquisition.

Programs which acquire and release storage dynamically cannot be totally optimized since some regions of storage are not static. One suggestion is to maintain a list of the available space in the program's pages and try to allocate dynamic regions in the same page as the sector which issued the storage request.

Another point of note is that execution of a program which has been processed by a linkage editor may not be as efficient as that of the program in unlinked form. Using a conventional linkage editor, page frames would be loaded with sequential program code regardless of any logical divisions. Thus, the likelihood of a loop within a sector crossing a page boundary is increased. A conflict now arises between the advantages and disadvantages of using a standard linkage editor. The advantage is faster program loading since most external references have been resolved when the program is processed by a linkage editor.

The implication is that a specialized linkage editor for paging systems is required. Its task would be to load pages with program modules while resolving external references; however, it would also attempt to prevent a module from crossing a page boundary. Efficient loading would minimize the amount of wasted space at the end of pages, and it is possible that program generated data areas could be allocated to these regions. A linkage editor of this type could also be fed information from automatic program monitors and thus perform the page loading on the basis of module interaction.

Even though there is sufficient indication to warrant development of a program monitor and reorganization system, there are still technical problems which must be overcome before implementing such a scheme. The algorithms and programs used in this experimental study are not considered to be useful in a practical sense. Therefore, a sufficiently sophisticated program monitor must first be developed.

Any monitoring that is performed must encompass numerous runs of the program to improve the chances that any inherent data dependencies are discovered. Improving the execution for the entire range of data which the program might receive, rather than just for one particular type of input, is the ultimate concern.

The results in this study concur with the findings of Hatfield [H2] in that large, modularly-designed programs such as the FORTRAN G compiler do not alter their behaviour significantly when the data is changed, or that changes in data result in sectors specific to the particular data variation being referenced. For example, if a syntax error is introduced into a program that previously compiled, one might find that the modules of the compiler which generate object code are now not referenced. Therefore, changing the data usually results in a different set of modules being accessed. It is vital that the program be tested with many data variations, in order to observe all sectors in execution. It will be interesting to see if any work in the area of dynamic determination of program locality is more successful than the static approach.

In addition to developing an efficient and easy-to-use monitor, a good reorganization algorithm must be produced. Following the failure to develop an algorithm for the optimal reorganization, a suboptimal heuristic solution with less than a specified error tolerance is needed. Acquiring the data necessary to verify real performance improvements may necessitate the development of a "page-fault" monitor to record the number of program interrupts caused by a missing-page exception. The results would have to be gathered in various system environments since the page-fault rate is a function of the number of pages which can be maintained in

main memory and this, in turn, is dependent on the workload.

Another experiment which could be performed to determine the effectiveness of the reorganization involves calculation of working-set sizes. Hopefully, such a scheme would reduce significantly the number of pages within the working sets of the program. If this goal is attained, systems employing a working-set page-replacement algorithm might be improved, since the working sets will be more clearly and compactly defined.

The type of reorganizational algorithms which have been mentioned in this study are based on the particular model of program behaviour developed in Chapter 3. Unfortunately, the problem which is being studied cannot be solved easily by such a rigorous mathematical model, since it requires the minimization of some function. Future work in this area should examine the possibility of incorporating time-series analysis of the memory reference string. It may well be that a repacking of program sectors according to the number of inter-sector references is not the best criterion and may not reduce actual page faults, even if there is significant reduction in potential page faults.

If programs do, in fact, maintain a set of pages in main storage during execution, then two sectors that are in separate pages but which reference each other a great deal

within a short time interval may well incur very little paging overhead, since their frequent contemporary use will stop them from being prime candidates for removal from main memory. However, two sectors which reference each other intermittently during the course of execution may be causing a page fault on each reference if they are in separate pages that are not in main memory at the same time. Therefore, much more intensive research on page-residence times and working-set sizes is a prerequisite to the development of more effective reorganizational algorithms.

The feasibility of a page-repacking system is closely related to the types of programs to which it is applied. It is recognized that the determination of program behaviour and subsequent page reorganization can be expensive; therefore, the system must be applied selectively. Prime candidates for possible reorganization include large, heavily used programs such as compilers, assemblers, text editors, etc.; paged portions of the operating system, especially the modules which are reentrant; large real-time applications whose sectors have widely differing frequencies of use; systems which have been divided into components that are programmed by different people. Also, interactive time-sharing systems require efficient organization of the software supporting the system in order to minimize response delays. Not all of these programs are good candidates and the monitor system should be capable of deciding whether or

not a program is eligible for reorganization.

There are several factors leading to increased popularity and use of paging systems which will in turn necessitate software either designed specifically for, or modified to perform efficiently in, these systems. One factor is IBM's announced main stream systems support, which includes the operating system OS/VS (Virtual Storage). Another factor that enhances the idea of page reorganization according to module interaction is the emphasis being placed on "structured" programming, which can only increase the number of major systems designed and programmed in modular form. Since in these and other systems there are usually several programmers working on separate sectors of the program, behaviour of the total program is even more clouded and can probably be best determined by some form of monitor.

There are already indications that the concepts discussed in this thesis are beginning to take practical form. An IBM Installation Newsletter [I6] reports the idea of tuning individual programs in order to lessen their "impact" on the operating system, namely OS/VS2 (Virtual Storage - Release 2). It is suggested that programming practices considered good for MVT are no longer acceptable for the paging operating system. The linkage editor for VS2 has been extended to allow the programmer to specify the order in which the CSECTS (sectors) are to be loaded.

Also, the programmer can dictate whether or not a CSECT is to begin on a page boundary.

It is concluded that the program reorganization scheme considered in this study can indeed be worthwhile provided that efficient monitor and reorganizational algorithms can be designed. The technique will be a valuable tool in paging systems; however, without the implementation of inexpensive algorithms to perform the reorganization, the concept may not progress beyond the manual reorganization suggested by Comeau [C5].

BIBLIOGRAPHY

[A1] Aho, A.V., P.J. Denning and J.D. Ullman, "Principles of Optimal Page Replacement", Journal of the ACM, Vol. 18, No. 1, Jan. 1971, pp. 80-92.

[A2] Alexander, M.T., Time Sharing Supervisor Programs, The University of Michigan Computing Centre, May 1969.

[A3] Arden, B.W. et al., "Programming and Addressing Structure in a Time Sharing Environment", Journal of the ACM, Vol. 13, No. 1, Jan. 1966, pp. 1-16.

[B1] Belady, L.A., "A Study of Replacement Algorithms for a Virtual Storage Computer", IBM Systems Journal, Vol. 5, No. 2, Feb. 1966, pp. 78-101.

[B2] Belady, L.A. and C.J. Kuehner, "Dynamic Space Sharing in Computer Systems", Communications of the ACM, Vol. 12, No. 5, May 1969, pp. 282-288.

[B3] Bovet, D.P., "Memory Allocation in Computer Systems", California University, June 1968, AD 670 499.

[B4] Brawn, B.S. and F.G. Gustavson, "Program Behaviour in a Paging Environment", Proceedings AFIPS 1968 FJCC, Vol. 33, pp. 1019-1032.

[C1] Coffman, E.G. and L.C. Varian, "Further experimental data on the behaviour of programs operating in a paging environment", Communications of the ACM, Vol. 11, No. 7, July 1968, pp. 471-474.

[C2] Corbato, F.J. et al., The Compatible Timesharing System, The MIT Press, Cambridge Mass., 1963.

[C3] Corbato, F.J., "A Paging Experiment with the Multics System", Rep. MAC-M-384, MIT Project MAC, Cambridge Mass., 1968, AD 687 552.

[C4] Control Data Star-1B, Hardware Reference Manual, 60326501.

[C5] Comeau, L.W., "A Study of the Effects of User Program Optimization in a Paging System", First Symposium on Operating System Principles, Oct. 1967.

[D1] Denning, P.J., "Virtual Memory", Computing Surveys, Vol. 2, No. 3, March 1970, pp. 153-189.

[D2] Denning, P.J., "The Working Set Model for Program Behaviour", Communications of the ACM, Vol. 11, No. 5, May 1968, pp. 323-333.

[D3] Denning, P.J., "Thrashing, Its Causes and Prevention", Proceedings AFIPS 1968 FJCC, Vol. 33, pp. 915-922.

[D4] Dennis, J.B., "Segmentation and the Design of Multiprogrammed Computer Systems", Journal of the ACM, Vol. 12, No. 4, April 1965, pp. 589-602.

[F1] Fine, G.H., C.W. Jackson and P.V. McIsaac, "Dynamic Program Behaviour under Paging", Proceedings 21st National Conference of the ACM, 1966, pp. 223-228.

[F2] Fotheringham J., "Dynamic Storage Allocation in the Atlas Computer", Communications of the ACM, Vol. 4, No. 10, Oct. 1961, pp. 435-436.

[G1] Garey, M.R., R.L. Graham and J.D. Ullman, "Worst Case Analysis of Memory Allocation Algorithms", Proceedings of the 4th Annual Symposium on the Theory of Computing, May 1972, pp. 143-150.

[H1] Hatfield, D.J., "Experiments on Page Size, Program Access Patterns and Virtual Memory Performance", IBM Journal of Research and Development, Vol. 16, No. 1, Jan. 1972, pp. 58-66.

[H2] Hatfield, D.J. and J. Gerald, "Program Restructuring Techniques for Virtual Memory", IBM Systems Journal, Vol. 10, No. 3, March 1971, pp. 168-192.

[I1] Iliffe, J.K., Basic Machine Principles, American Elsevier Publishing Co. Inc., New York, 1968, pp. 22-25.

[I2] Iliffe, J.K. and J.G. Jodeit, "A Dynamic Storage Allocation Scheme", Computer Journal, Vol. 5, 1963, pp. 200-209.

[I3] IBM System/360 Operating System MFT Guide, GC 27-6939.

[I4] IBM System/360 Operating System MVT Guide, GC 28-6720.

[I5] IBM System/360 Model 67 Functional Characteristics, GA 27-2719.

[I6] IBM Installation Newsletter, Issue No. 73-01, Jan. 26, 1973

[I6] Ingalls, D., "The Execution Time Profile as a Programming Tool", Design and Optimization of Compilers, Courant Computer Science Symposium 5, May 1971, Prentice Hall, pp. 107-128.

[J1] Jasik, S., "Monitoring Program Execution on the CDC 6000 Series Machines", Design and Optimization of Compilers, Courant Computer Science Symposium 5, May 1971, Prentice Hall, pp. 129-136.

[K1] Kilburn, T. et al., "One level storage system", IRE Transactions, EC-11, April 1962, pp. 223-235.

[K2] Kilburn, T., R.B. Payne and D.J. Howarth, "The Atlas Supervisor", Proceedings of the 1961 Eastern Joint Computer Conference, pp. 279-294.

[K3] Knuth, D.E., The Art of Computer Programming, Vol. 1, Addison Wesley, Reading Mass., 1968, pp. 435-455.

[K4] Knuth, D.E., "An Empirical Study of FORTRAN Programs", Software Practice and Experience, Vol. 1, No. 2, Feb. 1971, pp. 105-134.

[K5] Kernighan, B.W., "Optimal Segmentation Points for Programs", Second Symposium on Operating Systems Principles, Oct. 1968, pp. 47-53.

[K6] Kuehner, C. and B. Randell, "Demand Paging in Perspective", Proceedings AFIPS 1968 FJCC, Vol. 33, pp. 1011-1018.

[K7] Kinslow, H.A., "A timesharing monitor system", Proceedings AFIPS 1964 FJCC, Vol. 26, pp. 443-454.

[L1] Lowe, T.C., "Analysis of Boolean Models for Time-Shared Paged Environments", Communications of the ACM, Vol. 12, No. 4, April 1969, pp. 199-205.

[L2] Lowe, T.C., "Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Processor Timing", Communications of the ACM, Vol. 13, No. 1, Jan. 1970, pp. 3-6.

[M1] Mattson, R.L. et al., "Evaluation Techniques for Storage Hierarchies", IBM Systems Journal, Vol. 9, No. 2, Feb. 1970, pp. 78-117.

[M2] "*TALLY", Michigan Terminal System: Public File Descriptions, Vol. 2, Jan. 1972, p. 331.

[O1] O'Neill, R.W., "Experience using a timesharing, multiprogramming system with dynamic address relocation hardware", Proceedings AFIPS 1967 SJCC, Vol. 30, pp. 611-621.

[O2] Organick, E.I., The Multics System, The MIT Press, Cambridge Mass., 1972.

[O3] Oppenheimer, G. and N. Weizer, "Resource Management for a Medium Scale Time Sharing Operating System", Communications of the ACM, Vol. 11, No. 5, May 1968, pp. 313-322.

[P1] Pankhurst, R.J., "Program Overlay Techniques", Communications of the ACM, Vol. 11, No. 2, Feb. 1968, pp. 119-125.

[P2] Parmelee, R.P. et al., "Virtual Storage and Machine Concepts", IBM Systems Journal, Vol. 11, No. 2, Feb. 1972, pp. 99-130.

[P3] Peters, C.B., Experiments in Automatic Paging, Vol. 1, Informatics Inc., Nov. 1971, AD 734 253.

[R1] Randell, B. and C.J. Kuehner, "Dynamic Storage Allocation Schemes", Communications of the ACM, Vol. 11, No. 5, May 1968, pp. 297-305.

[R2] Randell, B., "A Note on Storage Fragmentation and Program Segmentation", Communications of the ACM, Vol. 12, No. 7, July 1969, pp. 365-369.

[R3] Ramamoorthy, C.V., "The analytic design of a dynamic lookahead and program segmenting system for multiprogrammed computers", Proceedings 21st National Conference of the ACM, 1966, pp. 229-239.

[S1] Sayre, D., "Is Automatic 'Folding' of Programs Efficient Enough to Displace Manual?", Communications of the ACM, Vol. 12, No. 12, Dec. 1969, pp. 656-660.

[S2] Salton, G., Automatic Information Organization and Retrieval, McGraw-Hill Book Co., 1968, pp. 135-139.

[T1] Thorington, J.M. and J.D. Irwin, Adaptive Replacement Algorithms for Use in Paged Memory Computer Systems, Project Themis: Information Processing, Technical Report AV-T-18, 1971.

[T2] Thorington, J.M. and J.D. Irwin, "An Adaptive Replacement Algorithm for Paged Memory Computer Systems", IEEE Transactions on Computers, Vol. C21, No. 10, Oct. 1972, pp. 1053-1061.

[V1] von Neumann, J., "First draft on a report on the Edvac", Rep. on Contract No. W-670-ORD-492, Moore School of Electrical Engineering, University of Pennsylvania, June 30, 1945.

[V2] Varian, L.C. and E.G. Coffman, "An Empirical Study of the Behaviour of Programs in a Paging Environment", First Symposium on Operating System Principles, Oct. 1967.

[V3] Ver Hoef, E., "Automatic program segmentation based on Boolean connectivity", Proceedings AFIPS 1971 SJCC, Vol. 39, pp. 491-495.

[W1] Winder, R.O., "A Data Base for Computer Performance Evaluation", Computer, Vol. 6, No. 3, March 1973, pp. 25-29.

[W2] Wilner, W.T., Design of the B1700, Burroughs Corp., Santa Barbara Plant, Goleta, Calif.

[W3] Wilner, W.T., B1700 Memory Utilization, Burroughs Corp, Santa Barbara Plant, Goleta, Calif.

APPENDIX

Sample Reference Matrix

Program 1a

The following table, which was referenced in Chapter 5, provides an illustration of the type of reference matrix which was gathered by the monitor program. The matrix shown here is for run "a" of program 1. The columns of the matrix extend over the following four pages. An individual numerical element, c(i,j), indicates the number of references made in sector "j" as a direct result of instructions executed in sector "i".

	ATTNTRP	<SYMTAB>	SISMAIN	FMTCMD	DISKLOCN	SISIO
ATTNTRP	31	0	1	0	0	0
<SYMTAB>	0	0	0	0	0	0
SISMAIN	4	1	88	3	0	5
FMTCMD	10	67	203	1560	0	31
DISKLOCN	0	4	0	0	184	18
SISIO	34	222	5	0	0	2568
FINDCAT	0	13	2	0	0	4
DEFNAME	3	23	0	0	0	8
AVE	0	165	0	0	0	2
COMPUTE	107	308	0	0	0	149
SELTERM	152	161	0	0	0	1179
COU	124	369	0	0	0	118
PRINT	1	0	0	0	0	373
SELECTF	1	0	0	0	0	19
LOOP	22	20	14	4	0	22
ALLOCATE	8	8	0	0	0	0
INTERUPT	14	11	7	0	0	2
PACKNUM	0	0	5	0	0	0
SUB	0	12	25	6	6	10
FIXNUM	0	0	0	0	30	0
FINDFD	0	0	20	0	0	0
OPEN	0	6	17	0	0	0
CLOSE	0	0	1	0	0	7
SCANNUM	0	0	20	0	0	0
SPIE	1	0	2	0	0	0
USERGET	0	0	0	0	0	0

	FINDCAT	DEFNAME	AVE	COMPUTE	SELTERM	COU	PRINT
ATTNTRP	0	0	0	0	0	0	0
<SYMTAB>	0	0	0	0	0	0	0
SISMAIN	0	1	0	0	7	1	0
FMTCMD	0	0	0	0	0	0	0
DISKLOCN	0	0	0	0	0	0	0
SISIO	0	0	0	0	0	0	0
FINDCAT	162	12	0	0	0	0	0
DEFNAME	2	176	8	0	0	6	0
AVE	0	1	773	1	1	0	0
COMPUTE	0	0	0	4586	152	0	0
SELTERM	0	0	0	912	9901	911	0
COU	0	1	0	166	153	2928	0
PRINT	0	0	0	256	0	0	1329
SELECTF	0	0	0	0	0	0	6
LOOP	1	0	1	4	0	4	4
ALLOCATE	0	0	0	0	0	0	0
INTERUPT	0	0	0	0	0	0	0
PACKNUM	0	0	0	0	0	0	0
SUB	1	0	0	0	0	0	0
FIXNUM	0	0	0	0	0	0	0
FINDPD	0	0	0	0	0	0	0
OPEN	0	0	0	0	0	0	0
CLOSE	0	0	0	0	0	0	0
SCANNUM	0	1	0	0	0	0	0
SPIE	0	0	0	0	0	0	0
USERGET	0	0	0	0	0	0	0

	SELECTF	LOOP	ALLOCATE	INTERUPT	PACKNUM	SUB
ATTNTRP	0	0	0	5	0	0
<SYMTAB>	0	0	0	0	0	0
SISMAIN	0	3	0	0	0	1
FMTCMD	0	24	0	0	0	36
DISKLOCN	0	0	0	0	0	35
SISIO	0	0	0	0	0	0
FINDCAT	0	6	0	0	0	6
DEFNAME	0	0	0	0	0	0
AVE	0	5	0	0	0	0
COMPUTE	0	21	0	0	0	0
SELTERM	0	0	0	0	0	0
COU	0	24	0	0	0	0
PRINT	1	24	0	0	0	0
SELECTF	62	0	0	0	0	0
LOOP	0	322	4	1	0	0
ALLOCATE	0	24	132	0	0	0
INTERUPT	0	5	0	151	0	0
PACKNUM	0	0	0	0	110	0
SUB	0	0	0	0	0	233
FIXNUM	0	0	0	0	5	0
FINDFD	0	0	0	0	0	0
OPEN	0	0	0	0	0	0
CLOSE	0	0	0	0	0	0
SCANNUM	0	0	0	0	5	0
SPIE	0	0	0	0	0	0
USERGET	0	0	0	0	0	0

	FIXNUM	FINDFD	OPEN	CLOSE	SCANNUM	SPIE	USERGET
ATTNTRP	0	0	0	0	0	2	0
<SYMTAB>	0	0	0	0	0	0	0
SISMAIN	0	1	1	0	0	1	0
FMTCMD	0	0	0	0	0	0	0
DISKLOCN	5	0	0	0	0	0	0
SISIO	0	0	0	1	0	0	596
FINDCAT	0	0	0	0	0	0	0
DEFNAME	0	0	0	0	0	0	0
AVE	0	0	0	0	0	0	0
COMPUTE	0	0	0	0	0	0	0
SELTERM	0	0	0	0	0	0	0
COU	0	0	0	0	0	0	6
PRINT	0	0	0	0	0	0	0
SELECTF	0	0	0	0	0	0	0
LOOP	0	0	0	0	0	0	0
ALLOCATE	0	0	0	0	0	0	0
INTERUPT	0	0	0	0	0	0	0
PACKNUM	30	0	0	0	5	0	0
SUB	0	0	0	0	0	0	0
FIXNUM	135	0	0	0	0	0	0
FINDFD	0	270	0	0	0	0	0
OPEN	0	0	316	0	0	0	0
CLOSE	0	0	1	32	0	0	0
SCANNUM	0	0	0	0	144	0	0
SPIE	0	0	0	0	0	24	0
USERGET	0	0	0	0	0	0	0